DISTRIBUTION DESIGN
IN
OBJECT ORIENTED DATABASES

A thesis presented in partial fulfilment of the requirements for the degree of

MASTER OF INFORMATION SCIENCE
IN
INFORMATION SYSTEMS

at Massey University, Palmerston North,
New Zealand

Hui Ma
2003

Abstract

The advanced development of object oriented database systems has attracted much research. However, very few of them contribute to the distribution design of object oriented databases. The main tasks of distribution design are fragmenting the database schema and allocating the fragments to different sites of a network. The aim of *fragmentation* and *allocation* is to improve the performance and increase the availability of a database system. Even though much research has been done on distributed databases, the research almost always refers to the relational data model (RDM). Very few efforts provide distribution design techniques for distributed object oriented databases.

The aim of this work is to generalise distribution design techniques from relational databases for object oriented databases. First, the characteristics of distributed databases in general and the techniques used for fragmentation and allocation for the RDM are reviewed. Then, fragmentation operations for a rather generic object oriented data model (OODM) are developed. As with the RDM, these operations include horizontal and vertical fragmentation. A third operation named *splitting* is also introduced for OODM. Finally, normal predicates are introduced for OODM. A heuristic procedure for horizontal fragmenting of OODBs is also presented. The adaption of horizontal fragmentation techniques for relational databases to object oriented databases is the main result of this work.

Acknowledgements

I would like to thank Professor Klaus-Dieter Schewe, my supervisor, for his patience, guidance, suggestions and constant support during this research. I am also thankful to Markus Kirchberg for his encouragement and guidance through the early stage of chaos and confusion. A special thanks goes to Madre Chrystal for her kindly devoting valuable time to proof read my draft.

The *Massey Masterate Scholarship*, which was awarded to me for the period February 2002 – February 2003 for graduate studies, was crucial to the successful completion of this project.

Finally, I am grateful to my husband and my parents for their patience and *love*. Without them this work would never have come into existence (literally).

Hui Ma
March 31, 2003

Table of Contents

Chapter 1

Introduction

Relational database systems have been well accepted because they reflect the nature of the structures of many organizations and enable the possibility of efficiently and effectively sharing the data. As computer-based systems have penetrated all areas, there are increased demands for the non-conventional applications, such as computer-aided design (CAD), geographic information systems, image, and graphic database systems, etc. However, the complex structure of the data in these applications cannot be adequately modelled by traditional relational databases. Consequently, object oriented databases are due to take the place of relational ones and become more and more popular. At the same time network facilities enable the distribution of database systems. However, approaches have been limited to the relational data model which lead to the emergence of distributed relational database management systems (DRDBMSs). But distributed object oriented database management systems (DOODBMSs) are due as well.

It is desirable to design a distributed object oriented database in such a way that the system can perform efficiently and effectively. The techniques of distribution design of relational databases have been intensely studied from the 1980s. There is also substantial research contributing to the study of the object oriented data model, but there is little research on the distribution design of object oriented databases.

The aim of this thesis is to study the existing (mature) distribution design techniques for relational databases and to adopt them into an object oriented approach. The following sections of this chapter will first review the definition of a distributed database and then study the reasons of the emergence of distributed database systems. Two main distribution

design techniques will also be briefly defined. After reviewing some research work that has been done in DRDBMSs we will set up the objective of studying fragmentation in DOODBMSs. Finally, the outline of this thesis will be presented.

1.1 What is a Distributed Database?

Ceri and Pelagatti [6] define a distributed database as a collection of data that logically belongs to the same system but is spread over the sites of a computer network. Özsu and Valduriez [24] give a similar definition: a distributed database system is a collection of multiple, logically interrelated databases distributed over a computer network. They explain that the logically related files, which are individually stored at each site of a computer network, are not enough to form a distributed database. There need to be a structure among them. Ceri and Pelagatti [6] support this view and state that the data at different sites must have properties that tie them together, and that access to the files should be via a common interface. They explain that physical distribution means that data does not reside at the same site in the same processor. Özsu and Valduriez [24] point out that physical distribution does not necessarily imply that the computer systems are geographically distributed. The sites among the network could even have the same address. They could be in the same room, but the communication between them is done over a network instead of shared memory, and the communication network is the only shared resource.

1.2 Why Distribution?

Ceri and Pelagatti [6] and Özsu and Valduriez [24] describe the motivation for the development of distributed databases. Distributed database research is motivated by the reliability, performance, and economic concerns of distributed databases in organizations. Reliability refers to the ability to tolerant faults. Performance refers to the ability to reduce query response time and increase throughput. Economic concerns relate to the reduction of data communication and update synchronization costs.

- Organizational Reasons

In recent years, the demand for more information by industries, governments and academic institutes has led to databases that have exceeded the physical limitations of centralized systems. The advances of telecommunication techniques make distributed database systems more affordable and useful [14, 1]. Ceri & Pelagatti [6] state that distributed databases are motivated by organizational reasons: many organizations, especially global organizations are often decentralized. For such organizations, implementing the information system in a decentralized way might be more suitable.

- Economic Reasons

  Ceri & Pelagatti [6] state that economic reasons are another motivation for the development of distributed databases. They argue that large, centralized computer centres are becoming questionable with respect to economies of scale. Özsu & Valduriez [24] support this view and state that it normally costs much less to put together a system of smaller computers with the equivalent power of a single big machine due to the advance of minicomputers and microcomputers. The authors also state that the communication cost can be reduced when distributed databases are implemented. If databases are geographically dispersed and the application accessing them are at the intersection of dispersed data, it will be much more economical to partition the relations and the applications so that the data processing can be done locally at each site.

- Reliability and Availability

  Improved reliability and availability is one of the potential advantages of distributed databases which the centralized databases lack [6, 24]. When replications of data have been placed at different sites, the crash of one site or the failure of the communication link would not necessarily make the data impossible to reach. When the system crashes and the communication link fails, even though some of the data will not be accessible, the distributed database system still provides limited services.

- Interoperability of Existing Databases

  Ceri & Pelagatti [6] also mention that when there are several databases already existing in an organization and there is the necessity of executing global applications,

distributed databases are the natural solution. In this case, the distributed database is designed bottom-up from existing local databases. These local databases may need to be reconstructed to some degree, but it is much cheaper than building a new integrated distributed database.

- Expandability

  Özsu & Valduriez [24] and Ceri & Pelagatti [6] state that it is easier to accommodate increasing database sizes in a distributed environment. If an organization grows by adding new and relatively autonomous branches or warehouses, then the distributed database approach supports the information needs of the new sites with the minimum degree of impact on the existing system.

- Local Autonomy

  Local autonomy is emphasized by Ceri & Pelagatti [6] as a major reason that many business organizations consider a distributed information system. Since data is distributed, a group of users that commonly share such data can have this data placed at the site they work. Thus, the local controls are allocated to local users to enable them to take partial responsibility for information management in the distributed database.

1.3 Distribution Design: Fragmentation and Allocation

Distribution design is one of the major research problems whose solution will enhance performance of the distributed databases. It involves data acquisition, fragmentation of databases, allocation and replication of the fragments, and local optimization. Fragmentation and allocation are the most important elements of a distributed database design phase. They play important roles in the development of a cost efficient system [24].

Fragmentation is a design technique to divide a single database into two or more partitions such that the combination of the partitions yields the original database without any loss or addition of information [25]. This reduces the amount of irrelevant data accessed by the application, thus reducing the number of disk accesses. The result of the fragmentation process is a set of fragments defined by a fragmentation schema. Fragmentation can be either horizontal, vertical or mixed.

Horizontal fragmentation partitions a relation or a class into disjoint unions (fragments), which will have exactly the same structure but different contents. Thus a horizontal fragment of a relation or class contains a subset of the whole relation or class instance. *Vertical fragmentation* results in attributes and methods being partitioned into different fragments and therefore reduces irrelevant data accessed by applications [28].

Allocation is the process of assigning a node on the network to each fragment after the database has been properly fragmented [24]. When data is allocated, it may either be replicated or maintained as a single copy. The replication of fragments will improve the reliability and efficiency of read-only queries. The intention of allocation is to minimize the data transfer cost and the number of messages needed to process a given set of applications, so that the system functions effectively and efficiently [17, 33, 24]. For the sake of simpleness, we will not consider replication of fragments when we discuss fragmentation in this thesis.

1.4 Objective: Fragmentation in Object Oriented Databases

The techniques of fragmenting relational databases have been intensely studied since the early 1980s. There are many different approaches to fragmentation and allocation for distribution design in relational databases. Navathe & Ra [22] develop a vertical partitioning algorithm using a graphical technique. Navathe, Karlapalem & Ra [21] propose a mixed fragmentation methodology for distributed database design. Tamhankar & Ram [33] propose an integrated methodology for fragmentation and allocation. Chu [8] designs two methods for partitioning attribute which treat the transaction as the decision variable.

Even though much research has been dedicated to the issue of object-oriented databases, little has been related to the distribution design of object oriented databases. The fragmentation of object oriented databases is a complex and still open research problem because the semantic model of the object oriented data model is much richer and more complicated than that of the relational model. The object oriented data model allows not only record constructors to be used but also set and other bulk type constructors. These constructions may even appear not only on the class level but also in nested structures [28]. The aim of this thesis is to generate distribution design techniques from traditional RDM to the OODM.

This thesis will concentrate on fragmentation, especially on horizontal fragmentation for OODM.

1.5 The Outline of the Thesis

The thesis will start with a brief review of the issues related to distribution design in relational databases in Chapter 2. The general characteristics of distributed databases will be reviewed. Some key concepts related to distributed databases and a framework for distribution design will also be presented.

Chapter 3 covers characteristics of distribution design in relational databases. Fragmentation and allocation in RDM are studied in this chapter. Also, an overview of related works is provided in this chapter.

In Chapter 4, the fundamentals of the object oriented data model will be discussed. At the same time, an example of the object oriented database schema will be presented in this chapter.

Chapter 5 will concentrate on some fragmentation techniques that can be used in the distribution design of object oriented databases. Splitting, horizontal fragmentation and vertical fragmentation will be presented and analyzed. Some design strategies will also be presented. Finally, related works will be reviewed.

Chapter 6 will introduce normal predicates in the object oriented data model. Then some heuristics of horizontal fragmentation in object oriented databases will be presented.

Finally, Chapter 7 contains the conclusion of this work. Future work is also listed in this chapter.

Chapter 2

Distributed Databases

Distributed Database Management Systems (DDBMSs) are characterized by several levels of transparencies that they can support. The classification of distributed databases is based on autonomy and heterogeneity. Distribution design is performed under a framework that sets the objective of the design. There are two approaches to the design of distribution. They constitute different approaches to the design process. But both of them might be applied to complement one another [24]. This chapter defines the fundamental concepts and sets the framework for discussing distributed databases. We start by reviewing the characteristics of a distributed database system. Then we will present a architectural model for distributed DBMSs. Two alternative design strategies will also be discussed in this chapter.

2.1 Characteristics

Some desirable functions that should be supported by a true distributed DBMS are proposed by Özsu & Valduriez [24]. Data independence is considered to be one of the main motivations for introducing databases. In a distributed database, data independence has the same importance as in traditional databases. However, a new aspect is added to the usual notion of data independence, namely distribution transparency [6]. Atre & Advisor [3] explain that data access should be transparent and synchronized to preserve database integrity. Özsu & Valduriez [24] define transparency as separation of the higher-level semantics of a system from lower-level implementation issues. In other words, transparency means that when the users are accessing the data, they do not need to know where the data is stored, in what

format it is stored, or how it is to be accessed [3].

In a distributed relational database environment, each relation can be partitioned into a set of fragments on the basis of relevant informational content. The fragments of the relation are stored at different sites. Furthermore, it might be preferable to duplicate some of this data at other sites for performance and reliability reasons [24]. The result is a distributed database which is fragmented and replicated. The database users would see a logically integrated, single image database even though it may be physically distributed. The DDBMS should enable users to access the distributed database as if it were a centralized one. The ideal form with full transparency would imply that a query language interface to a distributed DBMS is not different from a query interface to a centralized DBMS.

An ideal DBMS should provide a number of different types of transparencies. In the following sections the different levels of transparency will be reviewed.

2.1.1 Data Independence

Data independence is a fundamental form of transparency that centralized database systems can provide. Data independence refers to the fact that the definition and maintenance of data are independent from the applications and are controlled by a server of the DBMS [23]. In other words, data independence means that the actual organization of data is transparent to the application programmer. Programs are written with a conceptual schema [6]. Özsu & Valduriez [24]propose two types of data independence:

- Logical data independence refers to the immunity of user applications to changes in the logical structure of the database. For example, if a user application operates on a subset of the attributes of a relation, it should not be affected later when new attributes are added to the same relation.

- Physical data independence refers to hiding the details of the storage structure from user applications. Programmers should not be concerned with the details of physical data organization when they design a user application. And the user application should not need to be modified when data organizational changes occur with respect

to what data type the data is assigned and what storage hierarchies the data is distributed across.

2.1.2 Network Transparency (Distribution Transparency)

Network transparency or distribution transparency means that programs can be written as if the database were not distributed [6]. Özsu & Valduriez [24] state that there should be no difference between database applications that would run on a distributed database and those that would run on a centralized one. They also state that the user should be protected from operational details of the network and even the existence of the network. They separate the distribution transparency into location transparency and naming transparency.

- Location transparency means that the command used to perform a task is independent of both the location of the data and the system on which an operation is carried out.
- Naming transparency refers to the fact that a unique name is provided for each object in the database. If there is no naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

2.1.3 Replication Transparency

Özsu & Valduriez [24] state that replication transparency is dealing with whether the users should be aware of the existence of copies, or whether the system should handle the management of copies. In a DBMS with replication transparency, users should act as if there is single copy of the data. They also emphasize that replication transparency refers only to the existence of replicas not to the location.

Ceri & Pelagatti [6] explain that there are several reasons for considering data redundancy as a desirable characteristic: first, the locality of applications can be increased if the data is replicated at all sites where applications need it; and second, the availability of the system can be increased because one site failure does not stop the execution of applications at other sites if the data stored is replicated. The performance can be increased through replication, as the retrieval can execute on any copy of the data if there is more than one copy of the

data. But on the other hand, replication causes problems in updating databases. Thus, if the user application is not retrieval oriented but update oriented, it might not be a good idea to have too many copies of the data. Whether or not to have copies and how many copies to have is decided to a considerable degree by the nature of user applications [24].

2.1.4 Fragmentation Transparency

With fragmentation transparency, users are not aware of the data separation. Fragmentation transparency is implied if the database systems have the function of physical data independence. Fragmentation is often used to improve performance, availability and reliability. Fragmentation can also reduce the negative effects of replication because by partitioning the data, only a subset of the relation, not the full relation, needs to be stored. Less space is required and fewer data items need to be managed [24].

2.2 Key Concepts

First, in this section, we will review and explain basic concepts that include *heterogeneity*, *autonomy*, and *distribution* which are used in the classification of distributed database systems. Then, classification of distributed database systems: tight integration, multidatabase, and semiautonomous database systems will be discussed explicitly.

2.2.1 Heterogeneity

Heterogeneity in DDBMSs can arise at different levels in the system, including hardware at different sites, different operating systems, different network protocols, different local DBMSs, and different models that local DBMSs are based on [4, 6, 24]. However, an important distinction is at the level of local DBMSs and the model they are based on, because differences at lower levels are managed by the communication software not by the DBMS. Therefore, only the heterogeneity of DBMS, the model of DBMS and the semantics of the data model need to be considered. Hence, the term "homogeneous DDBMS" refers to a DDBMS with the same DBMS based on the same data model with the same semantics.

2.2.2 Autonomy

Autonomy is concerned with the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently [6, 24]. Autonomy refers to a function of a number of factors such as whether the component systems exchange information, whether they can independently execute transactions and whether each site is allowed to modify itself.

Requirements of an autonomous system have been specified in a variety of ways. Bell & Grimson [4] specify the dimensions of autonomy as:

- Design autonomy: local sites are given the freedom to decide the information content of the DB, to select the data model and to choose storage structures.
- Participation autonomy: local sites have the right to decide what data to contribute and when, and the freedom to decide when to come and to go.
- Communication autonomy: local sites have the right to decide how and under what terms to communicate with other sites in the network.
- Execution autonomy: local sites have the freedom to make decisions whether and how to process local operations to store, retrieve and update local data.

2.2.3 Distribution

In Özsu & Valduriez [24] distribution refers to the physical distribution of data over multiple sites. There are two alternative classes that DBMSs use to distribute data: client/server distribution and peer-to-peer distribution. Client/server distribution concentrates data management duties with servers, while clients focus on providing the application environment that includes the user interface [3]. The client and servers share the communication responsibility. The sites on a network are distinguished as "clients" and "servers", and their functionality is different [24]. In peer-to-peer systems, there is no distinction between client machines and servers. Each machine has full DBMS functionality and machines can communicate with each other to execute queries and transactions. Peer-to-peer systems are also called fully distributed systems [24].

2.2.4 Classification of Distributed DBMS

Bell & Grimson [4] divide distributed database management systems into homogeneous DDBMSs and heterogeneous DDBMSs. Homogeneous DDBMSs can be further divided into classes depending on whether or not they are autonomous. Heterogeneous DDBMSs can be further divided into classes based on wether they are integrated or not. The authors also categorize systems according to the degree of heterogeneity, the method of data distribution and the extent of local autonomy. Özsu & Valduriez [24] and Ceri & Pelagatti [6] present a working classification of possible design alternatives along three similar dimensions: autonomy, distribution, and heterogeneity. Özsu & Valduriez [24] propose a classification of the implementation alternatives of distributed database systems which is based on autonomy and heterogeneity. This is illustrated in Figure 2.1. The different implementation alternatives that cover the important aspects of these features of distributed database design are reviewed below.

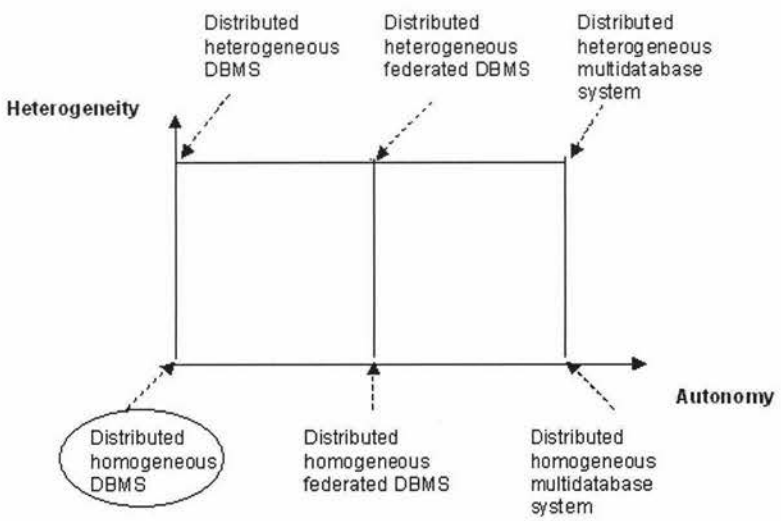


Figure 2.1: DDBMS Implementation Alternatives [24]

At one extreme, a tightly integrated (or truly distributed) database is a DDBMS with no local autonomy. In other words it has full global control. This kind of DDBMS has a global schema and all applications access the global database. The local DBMSs do not operate independently.

In contrast, in a multidatabase there are only local database schema and no global schema.

Global applications must access each local database separately. The individual systems are stand-alone DBMSs, which know neither of the existence of other DBMSs nor how to communicate with them. Local autonomy guarantees that local users can access their own local DB independently of the existence of the multidatabase and its global users.

Gligor & Popescu-Zeletin [15] list the requirements of multidatabase systems as follows:

- The local operations of the individual DBMSs are not affected by their participation in the multidatabase system.
- System consistency or operation should not be affected when a DBMS join or leave the multidatabase confederation.
- The manner of query processing and optimization of individual DBMS should not be affected by the execution of global queries that access multiple databases.

Semiautonomous (or federated) systems consist of a local database schema and a global schema. Local databases have their own schema and can operate independently, but they also have decided to participate in a federation to make their local data sharable. Each of these DBMSs determines what information it wants to share with other users. Applications may be executed locally and globally. They are not fully autonomous systems because the parts of their own databases that they share with other users should be able to be modified by other users in order to change information.

In this thesis, the discussion of data fragmentation and allocation is limited to the distributed homogeneous DBMS, which provides an integrated view of the data to users even though the database is distributed.

2.3 A Framework for Distributed Databases

The objective of the design of distribution is discussed by Ceri & Pelagatti [6]. The design of a distributed database system involves making decisions on the architecture of DDBMS, on the process of design. Two major strategies for designing distributed databases that are identified by Ceri & Pelagatti [6] are: top-down approach and bottom-up approach. Özsu

& Valduriez [24] develop a framework for the process of distribution design based on these approaches. The reasons and criteria for fragmentation and allocation will be discussed in this section.

2.3.1 The Objective of the Design of Data Distribution

There are several objectives that should be taken into account in the design of distribution [6]:

- In a distributed database system one of the major costs is associated with communication. To minimize communication costs, one goal of DDBMSs is to achieve processing applications locally. The degree of local processing can be maximized by distributing data, therefore minimizing transaction costs. To achieve this goal, the data should be kept as close as possible to the applications which use them. The advantage of processing applications locally is not only the reduction of remote access costs, but also increased simplicity in controlling the execution of the application.

- We can improve the availability and fault-tolerance of read-only applications by storing multiple copies of the same information at different sites. When one site of the database is down or the community link for that site is broken, the system can still execute the applications by accessing the other copies of the information.

- Distributing workload over the sites is done in order to take advantage of the different powers of utilization of the computers at each site, and to maximize the degree of parallelism of execution of applications. But the trade-off between processing locally and distributing workloads should be considered in the design of data distribution.

- Database distribution should reflect the cost and availability of storage at each site. Even though the storage cost is not relevant when compared with the cost of input or output (I/O), central processing unit (CPU), and transmission costs of the applications, the limitation of available storage at each site should be considered.

2.3.2 The Reasons for Fragmentation

To simplify the problem, we do not consider replication at the first step of distribution design. The purpose of fragmentation design is to determine non-overlapping fragments which could be the logical unit of allocation [6]. The individual tuple or attribute of a relation cannot be considered as the unit of allocation because the allocation problem would become unmanageable. The fragments are constituted by grouping tuples or attributes that have the same "properties" from the viewpoint of their application [6]. This is based on the idea that two elements in the same fragment that have the same "properties" will be accessed by the applications together. Therefore, the fragments obtained in this way are the appropriate units of allocation [6].

Özsu & Valduriez [24] mention that with respect to fragmentation, the important issue is the appropriate unit of allocation. The authors explain that there are three reasons for fragment relations:

- Applications are usually based on the views of subsets of relations. Thus the applications often access any subset of an entire relation locally.
- If there is a relation on which many application views are defined at different sites, storing a given relation at one site will result in an unnecessarily high volume of remote data accesses. Storing a given relation at different sites will cause problems in executing updates and may not be desirable if storage is limited.
- The decomposition of a relation into fragments permits many transactions to be executed concurrently and results in the parallel execution of a single query by dividing it into a set of sub queries that operate on fragments.

However, on the other hand, fragmentation may cause the following problems [24]:

- The applications whose views are defined on more than one fragment may suffer performance degradation when the relations are not partitioned into mutually exclusive fragments.

- When the attributes participating in a dependency of a relation are decomposed into different fragments and stored at different sites, the task of checking for dependencies would result in chasing after data in a number of sites.

2.3.3 Alternative Design Strategies

Ceri & Pelagatti [6] and Özsu & Valduriez [24] propose two alternative approaches to the design of data distribution, top-down and bottom-up approaches. In the case of tightly integrated distributed databases, design proceeds top-down from requirements analysis and logical design of the global database to physical design of each local database. In the case of distributed multidatabase systems, the design process is bottom-up and involves the integration of existing databases [24]. But the authors also emphasis the fact that real applications are rarely simple enough to fit nicely in either of these approaches. The two approaches may need to be applied to complement each other.

Top-down Approach

As shown in Figure 2.2, in the top-down approach, the process starts with a requirements analysis that defines the environment of the system and elicits both the data and processing needs of all potential database users [35]. The requirements analysis also specifies where the final system is expected to stand with respect to the objectives of the DDBMS. The objective is defined with respect to performance, reliability and availability, economics, and expandability (flexibility).

The requirements documents are the inputs to two parallel activities: view design and conceptual design. The outputs of view design are the interfaces for the user, and the outputs of conceptual design are entity types and relationship types which are used to construct an external schema.

In a distributed relational database environment, the objective of distribution design is to design the local conceptual schemas (LCSs) by distributing the relations and subrelations (fragments). The fundamental issues in top-down design are fragmentation and allocation [24].

The last step in the design process is the physical design, during which local conceptual schemas are mapped to the physical storage devices available at the corresponding local sites.

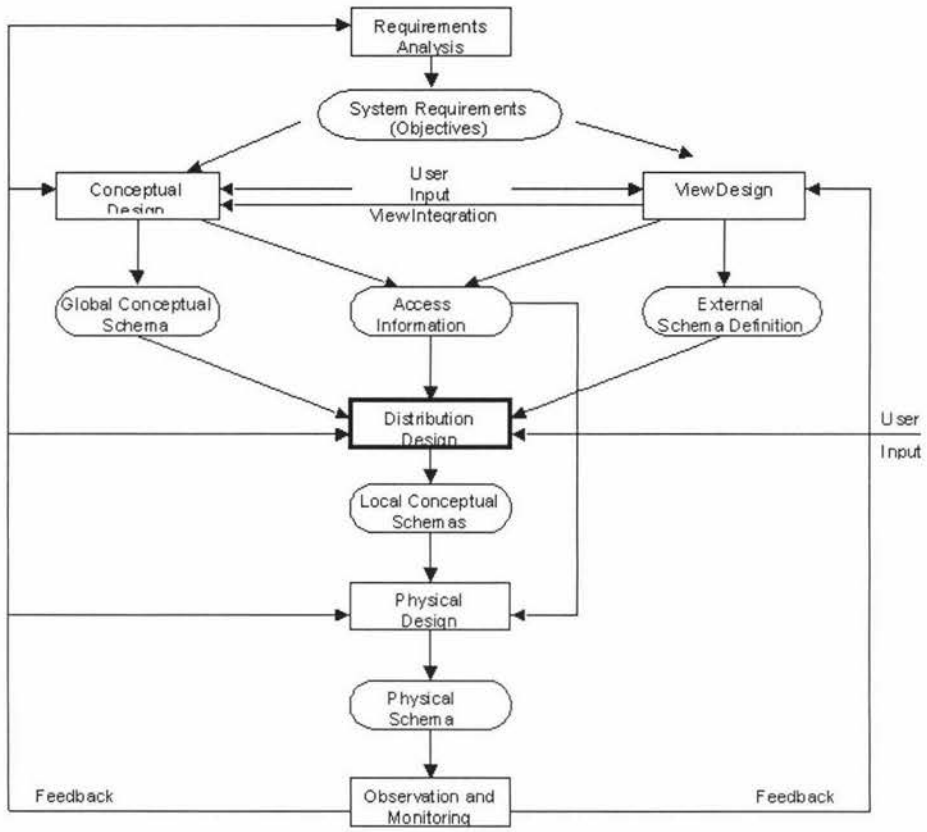


Figure 2.2: Top-down Design Process [24]

Bottom-up Design Process

Ceri & Pelagatti [6] and Özsu & Valduriez [24] state that top-down design is suitable for the systems which are developed from scratch. But when the distributed database is developed as the aggregation of existing databases, it is not easy to follow the top-down approach. The bottom-up approach, which starts with individual local conceptual schemata, is more suitable for this environment [6, 24]. Ceri & Pelagatti [6] explain that the bottom-up approach is based on the integration of existing schemata into a single, global schema. Integration is the process of the merging of common data definitions and the resolution of conflicts among different representations that are given to the same data. The global

conceptual schema is the product of the process [24]. Ceri & Pelagatti [6] conclude that there are three requirements for bottom-up design:

(1) the selection of a common database model for describing the global schema of the database,

(2) the translation of each local schema into the common data model, and

(3) the integration of the local schemata into a common global schema. The authors also state that these three requirements are particularly important in heterogeneous distributed systems.

Chapter 3

Distribution Design for Relational Databases

There are many reasons to study distributed relational databases. Firstly, the mathematical foundation of the relational data model (RDM) makes the theory research problem easy to formulate. Secondly, there are a large number of relational systems on the market, and most distributed database systems are relational [23]. Current distributed database management systems (DBMSs) are mainly available for relational databases (RDBs) [33].

The distribution design of databases involves data acquisition, fragmentation of the database, allocation and replication of the partitions and local optimization [19]. In a distributed relational database environment, fragmentation is the design technique that divides a relation into a set of partitions such that the combination of the partitions yields the original database without any loss or addition of information [24, 25].

Fragmentation can be done in several ways: horizontal, vertical, and mixed (hybrid). Horizontal fragmentation splits a relation into a set of disjoint new relations with the same relation schema. Each of the new relations is obtained by applying a selection operation to the original relation [28]. Vertical fragmentation involves dividing the attributes of a relation into groups and then applying a projection operation to the original relation over each group. The original relation can be reconstructed by union or join operations, respectively, of the new relations resulting from horizontal or vertical fragmentation [28].

The following sections will first set the preliminaries of the discussion of distribution design of RDB. Then we will list the characteristics of the distribution design. Afterwards we will

review each design technique mentioned above.

3.1 The Relational Data Model

The relational data model was first introduced by Codd [9] in 1970. Codd [10] emphasizes that the relational model has three power features. First, its data structures are simple. This feature allows a high degree of independence from the physical data representation. Second, the relational model provides a solid theoretical foundation for data consistency. The consistent states of a database can be uniformly defined and maintained through integrity rules. Thirdly, the relational model allows the set-oriented manipulation of relations. With this feature, powerful nonprocedural languages can be developed based either on set theory (relational algebra) or on logic (relational calculus).

To define the relational data model, we suppose that a certain set $\mathcal{D}$ of possible domains D is given. For instance, D can be NAT, which is a set of natural numbers, $STRING$, which is a set of character strings, and $BOOL$, that is a set of booleans, etc. Every attribute in a relation schema is assigned a domain D out of $\mathcal{D}$. The following definition is given in [28].

Definition 3.1. Let $\mathcal{D}$ be a non empty set of domains.

(i) A relation schema R consists of a finite set $attr(R)$ of attributes and a domain assignment $dom : attr(R) \rightarrow \mathcal{D}$.

(ii) A tuple over a relation schema R is a mapping $t : attr(R) \rightarrow \bigcup_{D \in \mathcal{D}}$ with $t(A) \in dom(A)$ for all $A \in attr(R)$.

(iii) A relation r over a relation schema R is a finite set of tuples over a relation schema R.

(iv) A relational database schema is a finite, non-empty set $\mathcal{S}$ of relation schemata.

(v) An instance $\mathcal{I}$ of a relational database schema $\mathcal{S}$ assigns to each $R \in \mathcal{S}$ a relation $\mathcal{I}(R)$ over R.

Remark:

(i) A relation schema R can be written in the form of $R = \{A_1 : D_1, \ldots, A_n : D_n\}$ with $attr(R) = \{A_1, \ldots, A_n\}$ and $dom(A_i) = D_i (i = 1, \ldots, n)$.

(ii) We usually use the term 'database' instead of talking about instances.

Each tuple in a relation is uniquely identified by its key which is the subset of the set of attributes $attr(R)$ of a relation r. The key is minimal if and only if there is no subset of a relation that can be a key. In order to define the key, we need to define a operation named projection first.

Definition 3.2. Let r be a relation over relation schema R, and $X \subseteq R$. For a tuple $t \in r$, projection of t, denoted as $t[X]$ is a mapping $t' : X \rightarrow \bigcup_{A \in X} dom(A)$ with $t' \in dom(A)$ such that $t'(A) = t(A)$ for each $A \in X$.

Then we can have the following definition:

Definition 3.3. A *key* on a relation schema R is a subset $K \subseteq attr(R)$ restricting relations r over R to satisfy $t_1 = t_2$ for all tuples $t_1, t_2 \in r$ with $t_1[K] = t_2[K]$.

A key K is called *minimal* if and only if no proper subset of K is a key.

A *primary key* is simply a distinguished minimal key

3.2 Characteristics

For a given relational database schema $\mathcal{S} = \{R_1, \ldots, R_k\}$, fragmentation applies to each of the relations R_i in the schema. To ensure the consistency of the database, the fragmentation operation should satisfy the following three characteristics [24]:

(i) **Completeness**

Informally, completeness refers to the fact that fragmentation does not lead to a loss of information. If a relation R_i is decomposed into fragments $R_i^1, \ldots, R_i^{ki}$, each data item that can be found in R_i can also be found in one or more of R_i^j.

(ii) **Reconstruction**

Informally, reconstruction refers to the fact that it should be easy to reconstitute a non-fragmented relation back from its fragments. For a given relation R_i with fragmentation $F_{R_i} = \{R_i^1, \ldots, R_i^{ki}\}$, it should be possible to define a relational operator that can reconstruct R_i with R_i^j.

- For horizontal fragments, reconstruction of a global relation R_i is performed by using the union operator in the horizontal fragmentation. Using relational algebra,

$$R_i = \bigcup_{j=1}^{k_i} R_i^j$$

- For vertical fragments, reconstruction of the original global relation R_i is made by using the natural join operation:

$$R_i = R_i^1 \bowtie \cdots \bowtie R_i^{k_i}$$

(iii) **Disjointness**

Informally, disjointness refers to the fact that fragmentation should not lead to a replication of information. But if replication comes into play, this requirement has to be relaxed. If a relation R_i is horizontally decomposed into fragments $R_i^1, \ldots R_i^{ki}$ and the data item d_i^n is in R_i^j, it does not appear in any other fragment $R_i^k (k \neq j)$. If a relation is vertically partitioned, disjointness is defined only on the non-primary key attributes of the relation. To simplify the problem, replication is not considered at the first stage of the design of distributed databases.

3.3 Horizontal Fragmentation

There are two forms of horizontal fragmentation: *primary horizontal fragmentation* and *derived horizontal fragmentation* [6, 24]. Schewe [27] defines primary horizontal fragmentation as:

Definition 3.4. Let $\mathcal{S} = R_1, \ldots, R_n$ be a relational database schema, *primary horizontal*

fragmentation on relation schema R_i replaces R_i by a set $\{R_i^1, \ldots, R_i^{ki}\}$ of new relation schemata such that:

(i) the attributes in each R_i^j are the same as in R_i.

(ii) Each relation r_i over R_i can be split into pairwise disjoint relations $r_i^1, \ldots, r_i^{ki}$ over $R_i^1, \ldots, R_i^{ki}$ respectively such that $r_i = r_i^1 \cup \cdots \cup r_i^{ki}$ holds.

By using relational algebra, the operation of horizontal fragmentation can be expressed as:

$$R_i^j = \sigma_{\varphi_j}(R_i)$$

with φ_j as selection predicates defined on R_i that are derived from application information. Of course, the disjointness property implies that we must have

$$\varphi_j \wedge \varphi_k \Leftrightarrow false \quad \text{for all} \quad j \neq k$$

Similarly, the completeness property implies the requirement that

$$\varphi_1 \vee \cdots \vee \varphi_{ki} \Leftrightarrow true$$

We notice the above definition includes the criteria which are also known as the correctness rules of horizontal fragmentation in the above section: *completeness, reconstruction*, and *disjointness.*

Example 3.1. Take relation schema LECTURER = {name, specialization, location}, a relation r over LECTURER is

name	specialization	location
Chris	Database Theory	Palmy
David	System Analysis	Wellington
Peter	End User Computing	Palmy
John	Database Theory	Auckland
Barbara	Multimedia	Auckland

with a set of selection formulae:

$$\varphi_1 \equiv \text{location} = \text{'Palmy'}$$

$$\varphi_2 \equiv \text{location} = \text{'Wellington'}$$

$$\varphi_3 \equiv \text{location} = \text{'Auckland'}$$

The above relation is partitioned into the following three relations:

name	specialization	location
Chris	Database Theory	Palmy
Peter	End User Computing	Palmy

name	specialization	location
David	System Analysis	Wellington

name	specialization	location
John	Database Theory	Auckland
Barbara	Multimedia	Auckland

□

Derived horizontal fragmentation is the partitioning of a relation that results from predicates being defined on another relation. For this operation, semi-joins: $R_1 \ltimes R_2 = \pi_{R_1}(R_1 \bowtie R_2)$ will be involved, we define derived horizontal fragmentation as below:

Definition 3.5. Let $\mathcal{S} = R_1, \ldots, R_i, R_x, \ldots, R_n$ be a relational database schema, *Derived Horizontal fragmentation* on relation schema R_i results that R_i be replaced by a set $\{R_i^1, \ldots, R_i^{ki}, R_i^{k_i+1}\}$ of new relation schemata such that:

(i) the attributes in each R_i^j are the same as in R_i,

(ii) each relation r_i over R_i can be split into pairwise disjoint relations $r_i^1, \ldots, r_i^{ki}, r_i^{ki+1}$ over $R_i^1, \ldots, R_i^{ki}, R_i^{ki+1}$ respectively such that $r_i = r_i^1 \cup \cdots \cup r_i^{ki} \cup r_i^{ki+1}$ holds.

Using relational algebra, the operation of derived horizontal fragmentation on relation R_i can be written as:

$$R_i^j = R_i \ltimes \sigma_{\varphi_j}(R_x)$$

with φ_j be the predicate defined on relation R_x and

$$R_i^{k_i+1} = R_i - \bigcup_{j=1}^{ki}(R_i \ltimes \sigma_{\varphi_j}(R_x))$$

be a necessary 'remainder fragment'.

Example 3.2. Take relation schemata LECTURER = {name, specialization, locationCode} and CAMPUS = {locationCode, city}, a relation r over LECTURE and relation r' over CAMPUS are:

name	specialization	locationCode
Chris	Database Theory	1
David	System Analysis	3
Peter	End User Computing	1
John	Database Theory	2
Barbara	Multimedia	2

locationCode	city
1	Palmy
2	Auckland
3	Wellington

with a set of selection formulae φ_j defined on relation CAMPUS, relation r over LECTURER can be fragmented as:

$$R_1 = \text{LECTURER} \ltimes \sigma_{\varphi_1 \equiv \text{city = 'Palmy'}}(\text{CAMPUS})$$

$$R_2 = \text{LECTURER} \ltimes \sigma_{\varphi_2 \equiv \text{city = 'Wellington'}}(\text{CAMPUS})$$

$$R_3 = \text{LECTURER} \ltimes \sigma_{\varphi_3 \equiv \text{city = 'Auckland'}}(\text{CAMPUS})$$

Then relation r is partitioned into the following three relations:

name	specialization	locationCode
Chris	Database Theory	1
Peter	End User Computing	1

name	specialization	locationCode
David	System Analysis	2

name	specialization	locationCode
John	Database Theory	3
Barbara	Multimedia	3

Note that the remainder relation r_4 is empty because all tuples of r can match a tuple in one of the fragment r'_j of relation r'. □

3.4 Vertical Fragmentation

Vertical fragmentation exploits relation schemata to be sets of attributes. Vertical fragments result from a projection operation to the original relation. The original relation can be reconstructed by the joining of the new relations. It can be defined as below:

Definition 3.6. Let $\mathcal{S} = R_1, \ldots, R_n$ be a relational database schema with relation schemata $R_i = \{A_{i1}, \ldots, A_{in_i}\}$. *Vertical fragmentation* replaces R_i by a set $\{R_i^1, \ldots, R_i^{ki}\}$ of new relation schemata such that:

(i) the attributes are distributed, i.e., $R_i = \bigcup_{j=1}^{k_i} R_i^j$,

(ii) each relation r_i over R_i is split into relations $r_i^j = \pi_{R_i^j(r_i)}(j = 1, \ldots, k_i)$ such that $r_i = r_i^1 \bowtie \cdots \bowtie r_i^{k_i}$ holds,

(iii) in Relational Algebra, $R_i = R_i^1 \bowtie \cdots \bowtie R_i^{k_i}$,

(iii') in a special case, a distinguishing new attribute dif can be added to relation R_i as a minimal key, then after vertical fragmentation $dif \in R_i^j$ for all $j \in \{1, \ldots, k_i\}$, and $R_i = \pi_{R_i - \{dif\}}(R_i^1 \bowtie \cdots \bowtie R_i^{k_i})$.

Using Relational Algebra, vertical fragmentation could be written as $R_i^j = \pi_{attr(R_i^j)}(R_i)$ for all $j \in \{1, \ldots, k_i\}$

Not having the distinguished new attribute dif would require a lossless join-decomposition, which in turn would mean that a join-dependency must hold. However, a well designed

database schema would exclude such dependencies, except for the case, where $R_i^1 \cap \cdots \cap R_i^{k_i}$ contains a key. Thus, it is normally required that the primary key (i.e. a chosen minimal key) is part of all R_i^j.

Example 3.3. Take relation schema LECTURER = {name, specialization, location, department, IRD, salary}, a relation r over LECTURER is

name	specialization	location	department	IRD	salary
Chris	Database Theory	Palmy	Information Systems	234569	92000
Richard	Accounting history	Wellington	Accounting	235169	38000
Peter	End User Computing	Palmy	Information Systems	256487	64560
Mary	Careers	Auckland	Human Resources	156426	65900
Barbara	Multimedia	Auckland	Computer Science	352486	56000

Relation r can be vertically fragmented in the following two ways:

(i) First, alter the relation schema LECTURER to LECTURER' by attaching a distinguishing new attribute 'dif' to LECTURER. Then, by using operation:

$$R_1 = \pi_{dif,\ \text{specialization, location, department}}(\text{LECTURER'})$$

$$R_2 = \pi_{dif,\ \text{name, IRD, salary}}(\text{LECTURER'})$$

the above relation r is vertically partitioned into the following two relations:

dif	specialization	location	department
1	Database Theory	Palmy	Information Systems
2	Accounting history	Wellington	Accounting
3	End User Computing	Palmy	Information Systems
4	Careers	Auckland	Human Resources
5	Multimedia	Auckland	Computer Science

dif	name	IRD	salary
1	Chris	234569	92000
2	Richard	235169	38000
3	Peter	256487	64560
4	Mary	156426	65900
5	Barbara	352486	56000

(ii) Alternatively, let {name, department} be the primary key. Using operation:

$$R_1 = \pi_{\text{name, department, specialization, location}}(\text{LECTURER})$$

$$R_2 = \pi_{\text{name, department, IRD, salary}}(\text{LECTURER})$$

the above relation r is vertically partitioned into the following two relations:

name	department	specialization	location
Chris	Information Systems	Database Theory	Palmy
Richard	Accounting	Accounting history	Wellington
Peter	Information Systems	End User Computing	Palmy
Mary	Human Resources	Careers	Auckland
Barbara	Computer Science	Multimedia	Auckland

name	department	IRD	salary
Chris	Information Systems	234569	92000
Richard	Accounting	235169	38000
Peter	Information Systems	256487	64560
Mary	Human Resources	156426	65900
Barbara	Computer Science	352486	56000

□

3.5 Mixed Fragmentation

Database users usually access subsets of data which are vertical and horizontal fragments of global relations. Therefore, there is a need for *mixed fragmentation*, which applies a sequence of horizontal and vertical fragmentation on a relation. It can be achieved by successively performing horizontal and vertical fragmentation on a relation. The different sequencing of vertical and horizontal operations generates different fragmentation schema. Even though these operations can be recursively repeated, having more than two levels of fragmentation is not of practical interest [6].

Definition 3.7. For a given relation R_i in the database schema $\mathcal{S}$, *mixed fragments* R_i^{jk} are built by successive steps of horizontal and vertical fragmentation on relation R_i, such that the correct rules for both vertical and horizontal fragmentation can be met. Using relational algebra, it can be expressed by using alternatively sequences of projection and selection operations:

$$R_i^k = \sigma_{\varphi_k}(\pi_{attr(R_i^k)}(\sigma_{\varphi_{k-1}}(\ldots(\sigma_{\varphi_1}(\pi_{attr(R_i^1)}(R_i))))))$$

allowing $\varphi_k \equiv true$ and $R_i^1 = R_i$ captures all other possibilities for such sequences.

3.6 Allocation

Once a fragmentation schema has been decided upon, each fragment must be assigned to one or more nodes in the distributed database management system. The allocation problem involves finding the "optimal" distribution of the fragments to the sites. The discussion of allocation is to find an allocation model that minimizes the total costs of processing and storage while trying to meet certain time restrictions [24]. The definition of a *cost minimized allocation* is as below:

Definition 3.8. For a given set of fragments $\{F_1, \ldots, F_n\}$ with different sizes $s_1, \ldots, s_n$, if the network has nodes $N_1, \ldots, N_k$, *fragment allocation* is to assign a node N_j to each fragment F_i such that the summary of all the transaction and storage costs from all the

sites can be kept to a minimum, where the transaction and storage costs are calculated according to a predefined cost model.

The focus of this thesis is on horizontal fragmentation of DOODBs. We will not go into detail on the discussion of vertical fragmentation and allocation of fragments.

3.7 Related Work

A lot of research has contributed to fragmentation and allocation in the area of distributed relational databases. This section will present some approaches for horizontal and vertical fragmentation.

3.7.1 Horizontal Fragmentation

For horizontal fragmentation, Ceri and Pelagatti [6] introduce two types of fragmentation: *primary* and *derived*. Derived fragmentation, which is performed to facilitate the join between fragments, is determined in terms of primary fragmentation.

- *Primary horizontal fragmentation* of a relation can be defined by determining a set of disjoint and complete selection predicates. Ceri and Pelagatti propose a procedure which produces a set of disjoint and complete selection predicates. This procedure can be summarized as follows:

  (i) Derive a set of simple predicates $P = \{p_1, \ldots, p_n\}$ from application information. Simple predicates take the form of:

  $$A_i = value$$

  where A_i is an attribute of the relation schema. The simple predicates $p_i \in P$ must satisfy two properties. They must be complete and minimal in order to guarantee that elements in the same fragments share the "same properties" in terms of allocation. The definition of complete and minimal can be found in [6].

(ii) Construct a set of *minterm predicates* from P by applying arbitrary conjunctions of all predicates p_i^*, where p_i^* is either $p_i \in P$ or its negation. Some of these conjunctions may be unsatisfiable. A set I of implications among the p_i^* can be used to determine (and remove) these unsatisfiable minterms.

The authors point out that it is not possible to analyse all the transactions that use the database. Only the most important and critical transactions should be taken into account. It is widely accepted that the "20/80" rule should be applied as a guideline to choose user applications to determine simple predicates. It means that only 20% of user queries should be taken into account, because they usually account for 80% of the data access. In particular, no update transactions will be considered. It is also suggested that fragments that have similar properties should not be distinguished. Otherwise the execution of the algorithms for a complete and minimal set of predicates will become very expensive.

- *Derived fragmentation* in [6] is performed by applying semijoin operations. Member relations are partitioned according to the fragmentation of their owners. By using relational algebra, derived fragmentation is expressed as $R_2^j = R_2 \ltimes R_1^j$ with R_2 indicating the member relations, R_1 indicating owner relations that have been fragmented into a set of disjoint fragments $\{R_1^1, \ldots, R_1^i\}$ with $1 \leq j \leq i$.

Öszu and Valduriez [24] follow the lines of Ceri, Pelagatti and Navathe [6, 20] and develop horizontal partitioning algorithms. Their fragmentation algorithm is based on the following necessary information requirements:

(i) **Database information.** The database information concerns the global conceptual schema. From the global conceptual schema, we can know how the database relations are connected to one another, especially with joins.

(ii) **Application information.** This includes the description of user queries and frequencies with which user applications access and update data. The quantitative and qualitative information acquired from application information can be summarized in the following four categories:

- **Simple predicates** for relation $R = \{A_1 : D_1, \ldots, A_n : D_n\}$ can be defined in the form of

$$p_j : A_i \theta Value$$

with A_i as an attribute defined over domain D_i, $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and $Value \in D_i$. A set of all simple predicates defined on relation R is denoted by $Pr = \{p_1, p_2, \ldots, p_m\}$.

- **Minterm predicates** are the conjunctions of simple predicates and their negations. The set of minterm predicates $M_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$ over a set Pr_i of simple predicates is defined by:

$$M_i = \{m_{ij} | m_{ij} = \bigwedge_{p_{ik} \in Pr_i} p_{ik}^*\}$$

where $p_{ik}^* = p_{ik}$ or $p_{ik}^* = \neg p_{ik}$. Note that all simple predicates in Pr_i appear (positively or negatively) in each minterm predicate.

- **Minterm selectivity** presented with $sel(m_i)$ is the number of tuples of the relation that would be accessed by a user query specified according to a given minterm predicate m_i.

- **Access frequency** with which users access data. For user application q_i, it is denoted with $acc(q_i)$. For a minterm predicate m_i it is represented as $acc(m_i)$.

The input of the algorithm for horizontal fragmentation is a relation R and a set of simple predicates Pr. The output of the algorithm is a set of fragments $\{R_1, \ldots, R_n\}$ of R. The objective of the algorithm is that Pr should be *complete* and *minimal*, which is defined as follows:

- **Completeness of simple predicates**

  A set of simple predicates P_r is said to be *complete* if and only if there is an equal probability of access by every application to two tuples belonging to the same minterm fragment that is defined according to P_r.

  For example, if we assume there is a relation PROJECT ={PNumber: *NAT*, PName: *STRING*, Budget: *NAT*, Campus: *STRING*, Description: *STRING*, Begin: *DATE*,

End: *DATE*}, there are two applications defined on it, and there are only three different locations for Project.

Find the budget of projects at each location. (a)

Find projects with budget less than $200000. (b)

According to application (a) Pr={Campus='Wel', Campus='PN', Campus='Auk'} is not complete with respect to application (b) because application (b) will access two tuples in a fragment defined by predicate Pr with different probability if the values of the budget of one object is more than or equal to $200000 and that of the other object is less than $200000. A complete set of simple predicates should be Pr={Campus='Wel', Campus='PN', Campus='Auk', Budget<200000, Budget $\geq$ 200000 }.

- **Minimality of simple predicates**

If a predicate influences how fragmentation is performed (e.g. f be partitioned into f_i and f_j), there should be at least one application that accesses f_i and f_j differently. In other words, the simple predicate should be *relevant* in determining a fragmentation. If all the predicates of a set P_r are relevant, then P_r is *minimal*.

If $p_i \in m_i$ and fragment f_i is determined by m_i, $\neg p_i \in m_j$ and fragment f_j is determined by m_j, then p_i is *relevant* iff

$$\frac{acc(m_i)}{card(f_i)} \neq \frac{acc(m_j)}{card(f_j)}$$

Where $acc(m_i)$ and $acc(m_j)$ denote the access frequencies of minterm predicate m_i and m_j, $card(f_i)$ and $card(f_j)$ denote the cardinalities of fragment f_i and f_j.

For instance, a set of simple predicates Pr ={Campus='Wel', Campus='PN', Campus='Auk', Budget< 200000, Budget $\geq$ 200000} is minimal in addition to being complete. However, if there is another predicate p_j ≡PName='BigNet' in Pr, then Pr ={Campus='Wel', Campus='PN', Campus='Auk', Budget<200000, Budget $\geq$ 200000, PName='BigNet'} is not minimal because there is no application that accesses fragment f_i (PName is 'BigNet') and f_j (PName is not 'BigNet') differently.

Özsu and Valduriez [24] first present an iterative algorithm named COM_MIN to generate a complete and minimal set of predicates Pr' from a given set of simple predicates Pr. The algorithm checks each predicate p_i in the given set of simple predicates Pr to see if it can be used to partition the relation R into at least two parts which are accessed differently by at least one application. If p_i satisfies the fundamental rule of completeness and minimality then it should be included in Pr'. If p_i is nonrelevant then it should be removed from Pr'. But this algorithm is not practical because checking p_i can not be defined with machine readable language. Moreover, it does not consider the fragment combination possibility that some of the minterm fragments might be allocated to the same site.

A algorithm named PHORIZONTAL is introduced to describe primary horizontal fragmentation. It uses the algorithm COM_MIN and a set of implications I as inputs to produce a set of satisfiable miniterm predicates M. If a minterm predicate m_i is contradictory to a implication rule in I, then it is removed from M. Minterm fragments are defined according to the set of satisfiable minterm predicates M. But the set I of implications is hard to define.

For the derived horizontal fragmentation, semijoin operations are involved. Member relations are fragmented according to the fragments of its owner relation. Details of the definition of member and owner relations can be found in [24]. They emphasize that care should be taken with the relations that have more than one link to the owner relations. Two criteria are suggested [24]. First, choose the fragmentation with better join characteristics. Second, choose the fragmentation used in more applications.

In fact, the algorithm is not very practical, as it will always result in a subset Pr' of Pr, the set of minterm predicates M' determined by Pr' and the corresponding set of fragments. Simple predicates are omitted from Pr if they do not contribute to the fragmentation, i.e. they only violate the minimality principle. This emerges to considering just the simple predicates $A_i\theta v_i$ in the most important queries and to take all satisfiable minterm predicates. This obviously leads to fragments that are accessed differently by at least two queries. The algorithm further does not give executable rules for eliminating the unsatisfiable minterm predicates.

The major problem, however, is that the number of fragments resulting from the algorithm is exponential in the size of Pr. In practice, it would be important to reduce this number significantly, which would mean to re-combine some of the fragments. In fact, this implies giving up the completeness principle and replacing it by optimization criteria based on a cost model.

Several other researchers have worked on fragmentation in the relational data model. Navathe *et al.*[21] define a schema for simultaneously applying the horizontal and vertical fragmentation algorithms on a relation to produce a grid. They use a technique similar to the vertical fragmentation presented in [20, 22] to produce horizontal fragments. The fragmentation schema generated by the algorithm is independent of the sequencing of the horizontal or vertical fragment algorithms. Tamhankar and Ram [33] introduce an integrated methodology for fragmentation and allocation. They make an attempt to combine fragmentation, allocation and replication into a single step of distribution design and apply the combination to a practical problem.

3.7.2 Vertical Fragmentation

Vertical fragmentation is more complicated than horizontal fragmentation because of the total number of alternatives. If there are n simple predicates that are used to define horizontal fragmentation, there are at most 2^n possible fragments that can be defined. But in the case of vertical fragmentation, if a relation has m nonprimary key attributes, the possible fragments are given by the Bell number which is approximately $B(m) \approx m^m$. From the value of the possible vertical fragmentation, we find out it is impossible to get the optimal solutions to the vertical partitioning problem. We can only expect to find out a heuristic solution.

There are several algorithms of vertical fragmentation that have been proposed in the literature. Hoffer and Severance [7] measure the affinity between pairs of attributes and try to cluster attributes according to their pairwise affinity by using the bond energy algorithm(BEA).

Navathe *et al.*[20] extend the BEA approach and proposed a two-phase approach for vertical

partitioning. In the first step, they used the given input parameters in the form of an attribute usage matrix(AUM) to construct the attribute affinity matrix (AAM) on which clustering is performed. In the second step, estimated cost factors, which reflect the physical environment of fragment storage, were considered to further refine the partitioning schema.

Cornell and Yu [11] apply the work of Navathe *et al.*[20] to physical design of relational databases. This approach uses specific physical factors such as the number of attributes, their length and selectivity, and cardinality of the relation.

Navathe and Ra [22] construct a graph-based algorithm to solve the vertical partitioning problem where the heuristics used include an intuitive objective function which is not explicitly quantified.

With the aim to overcome the complexity of attribute based algorithms, P.-C. Chu [8] proposes a transaction-oriented approach to vertical partitioning, in which no Attribute Utility Matrix but transaction information is used as the decision variable.

The algorithm presented in Özsu and Valduriez [24] takes AAF as input. The Bond Energy algorithm is employed to evaluate the togetherness of a pair of attributes. Shift operation is used to process the attribute clustering. The binary partitioning algorithm should be applied recursively when there is a large set of attributes.

Muthuraj *et al* [19] propose a formal approach to address the problem of an n-array vertical partitioning problem and derive a partition evaluator function which describes the affinity value for clusters of different sizes. This function can either be used for fragmentation progress or applied to evaluate the fragmentation schemata that are created and therefore to test and evaluate the different algorithms available. They argue that an attribute usage matrix instead of an attribute affinity matrix (AAM) should be used in the process of fragmentation because AAM can only measure the closeness of a pair of attributes at one time and cannot measure the closeness of the entire cluster which may consist of more than two attributes.

Both horizontal and vertical fragmentation problems are complex. Studying only one of them will be a hard task. This thesis will only concentrate on horizontal fragmentation and will not handle vertical fragmentation in detail.

Chapter 4

Object Oriented Databases

Object-oriented databases have brought about a fundamental change in the way a data and the procedures that operate on the data are viewed. Whilst general agreement on the broad features to be supported by an object oriented database system is slowly being reached, as yet there is still no firm agreement on a formal definition of an object-oriented database system [12, 30, 32]. Additionally, there is still much debate on which underlying object oriented data model is appropriate for database systems.

However, it is widely accepted that objects are abstractions of real world entities. Objects are regarded as the basic unit of persistent data [30], and the object oriented databases are composed of independent objects. A unique identifier should be assigned to an object because the existence of an object should be independent of its value [2, 30, 18]. The using of immutable object identifiers enable sharing, mutability of values and easily representation of cyclic structures.

Our research will apply the object oriented data model (OODM) which is presented in [30, 31]. This model applies an abstract object identifier to capture the fact that an object in the real world always has an unique identity. At the same time, an object in the real world can have different aspects and should not only be coupled with a unique type. In contrast, objects, as well as references to other objects, should be associated with more than one type that can change during the object's lifetime. The section below will present some fundamental concepts of the object oriented data model. It will also depict concepts by giving an object oriented database schema as well as an instance of the database.

4.1 Fundamentals of the OODM

This section will review the object model proposed in Schewe & Thalheim [30]. In this model each *object* o consists of a unique identifier id, a set of (type-, value-) pairs (T_i, v_i), a set of (reference-, object-) pairs (ref_i, o_j) and a set of methods $meth_k$. Values and objects are different in that values can be identified by themselves while objects can only be identified by applying an external identification mechanism. Types are used to structure values while classes are the groups of objects that have the same structure which uniformly combines aspects of object values and references. Subtyping is used to relate values with different types [31]. In this thesis, in order to simplify the problem of distribution design, the behavior(method) part will be omitted completely. The object oriented data model that we adapt to this project is just a simplified version of the data model introduced in [30, 31].

4.1.1 Type Definitions

The type system presented in [30, 31] consists of some *basic types*, *type constructors* and *subtyping* relation.

Definition 4.1. (i) The *base types* are either *BOOL*, *NAT*, *INT*, *FLOAT*, *STRING*, *ID*, or $\perp$, where *ID* is an abstract identifier type without any non-trivial supertype and $\perp$ is the trivial type that is a supertype for every type.

(ii) Let N_P and N_F denote parameter-names and function-names. Let $a_i \in N_F$ and $\alpha, \alpha_i \in N_P (i = 1, \ldots, n)$. A *type constructor* is either a record constructor $(a_1 : \alpha_1, \ldots, a_n : \alpha_n)$, a finite set constructor $\{\alpha\}$, a list constructor $[\alpha]$, a bag constructor $\langle\alpha\rangle$ or a union constructor $(a_1 : \alpha_1) \cup \cdots \cup (a_n : \alpha_n)$.

(iii) A type t is called *proper* iff the number of its parameters is 0. If there is no occurrence of *ID* in t, t is called a *value type*. If t' is a proper type occurring in a type t, then there exists a corresponding *occurrence relation* $o : t \times t' \rightarrow BOOL$.

New types can be defined by nesting using predefined base types, such as *BOOL, NAT, STRING*, etc, and predefined constructors for records, finite sets, lists, unions, etc. So we

can first define some types in the example below.

Example 4.1. First, we define *PERSONNAME* by using both a set constructor$\{\cdot\}$ and record constructor $(\cdot)$:

Type *PERSONNAME*

= (FName: *STRING*,

LName: *STRING*,

Title: $\{STRING\}$)

End *PERSONNAME*

Then we define PERSON based on type PERSONNAME.

Type *PERSON*

= (PersonID: *NAT*,

Name: *PERSONNAME*,

Address: *STRING*,

DOB: *DATE*)

End *PERSON*

We can also define the following types which will be used when we define a university database schema in the next subsection.

Type *PROJECT*

= (PNumber: *NAT*,

Pname: *STRING*,

Begin: *DATE*,

End: *DATE*,

Description: *STRING*

Budget: *STRING*)

End *PROJECT*

Type *COURSE*

= (CNumber: *NAT*,

Cname: *STRING*)

End *COURSE*

Type *DEPARTMENT*

= (Dname: *STRING*,

{(TelNumber: *NAT*,

Campus: *STRING*)})

End *DEPARTMENT*

Type *ROOM*

= (Building: *STRING*

NO: *NAT*

Campus: *STRING*)

End *ROOM*

Type *SEMESTER*

= (Year: *NAT*,

Season: *NAT*)

End *SEMESTER* □

Subtypes are also introduced in [30]. They are used to relate values in different types.

Definition 4.2. Let $\alpha_1, \ldots, \alpha_n$ be parameter-names. A *subtype relation* $\leq$ on types is given by the following rules:

(i) Every type t is its own subtype and a subtype of $\perp$.

(ii) $NAT \leq INT \leq FLOAT$.

(iii) $(\ldots, a_{i-1} : \alpha_{i-1}, a_i : \alpha_i, a_{i+1} : \alpha_{i+1}, \ldots) \leq (\ldots, a_{i-1} : \alpha'_{i-1}, a_{i+1} : \alpha'_{i+1}, \ldots)$ whenever $\alpha_j \leq \alpha'_j$.

(iv) $\left\{ \begin{array}{rcl} \{\alpha_i\} & \leq & \{\alpha_j\} \\ [\alpha_i] & \leq & [\alpha_j] \\ \langle\alpha\rangle & \leq & \langle\alpha_j\rangle \end{array} \right\}$ iff $\alpha_i \leq \alpha_j$.

(v) $\{\alpha\} \leq \langle\alpha\rangle$ and $[\alpha] \leq \langle\alpha\rangle$.

(vi) $\cdots \cup (a_{i-1} : \alpha_{i-1}) \cup (a_{i+1} : \alpha_{i+1}) \cup \cdots \leq \cdots \cup (a_{i-1} : \alpha'_{i-1}) \cup (a_i : \alpha'_i) \cup (a_{i+1} : \alpha'_{i+1}) \cup \ldots$ whenever $\alpha_i \leq \alpha'_j$ for all $j = 1, \ldots, n$.

Example 4.2. We can define *STUDENT* and *LECTURER* as subtypes of *PERSON* defined in the above example.

`Type` *STUDENT*

= (PersonID: *NAT*,

StudentID: *NAT*)

`End` *STUDENT*

`Type` *LECTURER*

= (PersonID: *NAT*,

Name: *PERSONNAME*,

Specialization: *STRING*)

`End` *LECTURER* □

4.1.2 Class Definitions

In the OODM presented in [30], classes are used to structure objects having the same structure and behavior, while types are used to structure values. Each object in a class has an identifier, a collection of values, references to other objects and methods. Identifiers can be represented by using the unique identifier type *ID*. An object, which has multiple aspects, can simultaneously belong to different classes. This property guarantees that each object of the abstract object model can be captured by the collection of possible classes. A class structure allows us to uniformly combine aspects of object values and references. Relationships between classes are represented by references and referential constraints on the object identifiers involved [30]. Because we are not looking at the dynamics, the model we apply in this thesis will not consider methods.

Definition 4.3. (i) Let t be a value type with parameters $\alpha_1, \ldots, \alpha_n$ such that *ID* does not occur in t. If the parameters are replaced by pairs $r_i : C_i$ with pairwise different reference names r_i and class names C_i, then the resulting expression is called a *structure expression.*

(ii) A class consists of a class name C, a structure expression exp_C, a set of class names $D_1, \ldots, D_m$ (called *superclasses*). The proper type derived from exp_C by replacing each reference $r_i : C_i$ with the type *ID* is called *representation type* T_C of the class C.

4.1.3 Schema Definition

The database schema is designed by a finite collection of type and class definitions [30]. Let us first review the definition of schema and the way to make it closed. Then we will look at an instance of a schema and some integrity constraints.

Definition 4.4. A *schema* $\mathcal{S}$ is a finite set of classes that is closed in that all names appearing in a structure expression or as superclass must be names of classes defined in the schema.

Definition 4.5. An *instance db* of a structure schema $\mathcal{S}$ assigns to each class $C \in \mathcal{S}$ a finite set $db(C)$ of values of type $(id : ID, value : T_C)$ such that the following conditions are satisfied:

uniqueness of identifiers: For every class C we have

$$\forall id :: ID.\forall v, w :: T_C.(id, v) \in db(C) \wedge (id, w) \in db(C) \Rightarrow v = w.$$

inclusion integrity: For a subclass C of C' we have

$$\forall id :: ID.id \in dom(db(C)) \Rightarrow id \in dom(db(C')).$$

Moreover, if T_C is subtype of T'_C with subtype function $f : T_C \rightarrow T'_C$, then we have

$$\forall id :: ID.\forall v :: T_C.(id, v) \in db(C) \Rightarrow (id, f(v)) \in db(C')$$

referential integrity: For each reference from C to C' with corresponding occurrence relation o_r we have

$$\forall id, id' :: ID.\forall v :: T_C.(id, v) \in db(C) \wedge o_r(v, id') \Rightarrow id' \in dom(db(C'))$$

where $dom(db(C)) = \{id :: ID | v :: T_C.(id, v) \in db(C)\}$.

4.2 An Example of Object Oriented Database Schema

In this section, we show an example of a database schema and its instance, the contents of the database at a given time point. Example 4.3 shows an object oriented database schema transformed from a Higher Order Entity Relationship Schema in [34]. Figure 4.1 is the Higher Order Entity Relationship Model (HERM) diagram of the schema.

We use the types defined in Example 4.1 to build the structural part $\mathcal{S}$ of the OODM schema. Therefore the structure of a class can be based on a type definition defined above or on a nameless type definition. It may also involve an IsA relation to model objects in more than one class. We use $\circ$ to indicate concatenation for record types.

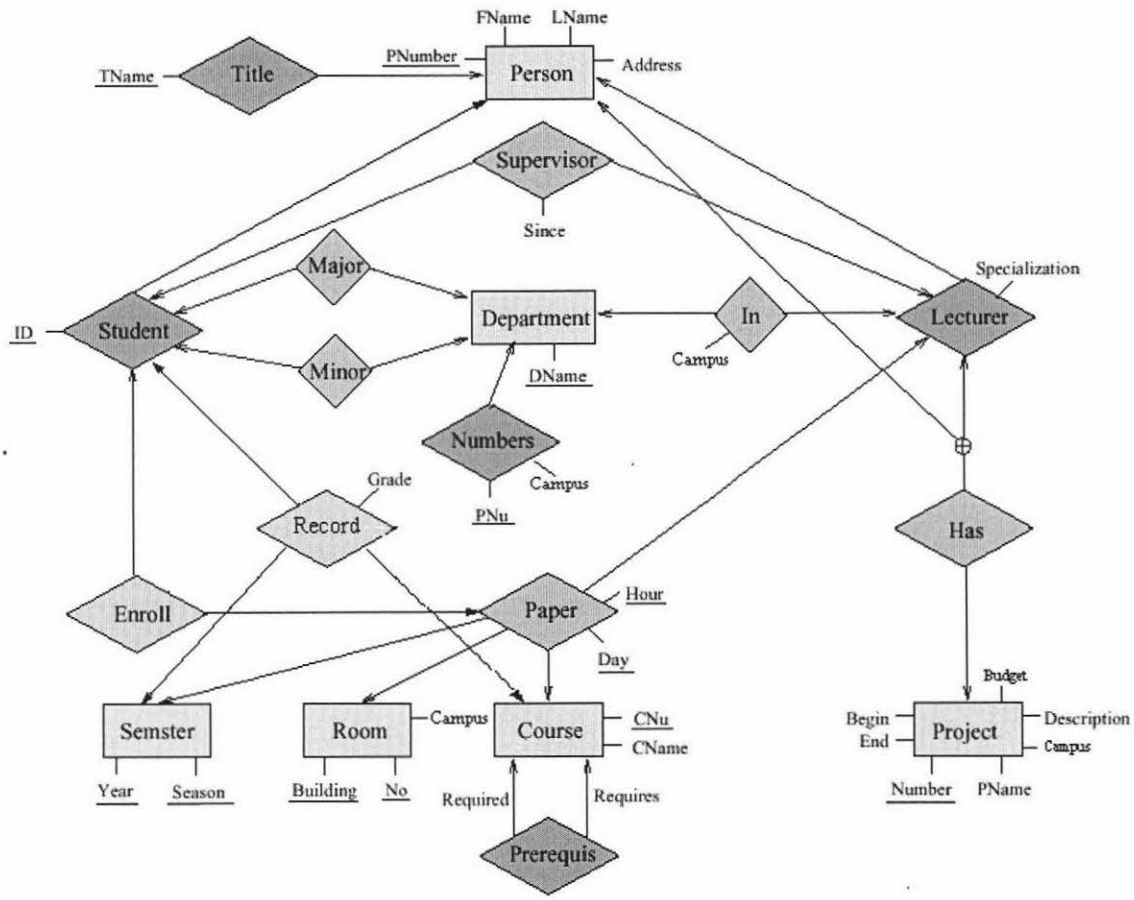


Figure 4.1: HERM Diagram of University Database Schema [34]

Example 4.3. We design a structural schema as below:

Schema University

Class PERSONC

Structure *PERSON*

End PERSONC

Class SEMESTERC

Structure *SEMESTER*

End SEMESTERC

Class ROOMC

Structure *ROOM*

End ROOMC

Class DEPARTMENTC

Structure *DEPARTMENT*

End DEPARTMENTC

Class COURSEC

Structure *COURSE* ∘

(Requires: {r: COURSEC})

End COURSEC

Class LECTURERC

IsA PERSONC

Structure *LECTURER* ∘

(Department: DEPARTMENTC,

Campus: *STRING*)

End LECTURERC

Class PROJECTC

Structure *PROJECT* ∘

(PrimaryInvestigator: { PERSONC ∪ LECTURERC })

End PROJECTC

Class PAPERC

Structure (Course: COURSEC ,

Semester: SEMESTERC,

Campus: *STRING*,

Lecturer: LECTURERC,

{(Day: *DATE*,

Hour: *NAT*,

Room: ROOMC)})

End PAPERC

Class STUDENTC

IsA PERSONC

Structure *STUDENT* ∘

(Supervisor: LECTURERC,

Major: DEPARTMENTC,

Minor: DEPARTMENTC,

Enroll: {(PAPERC,

Grade: *STRING*)})

End STUDENTC □

After we define the structure of the schema, we are going to describe the instances of the above schema. The instance of a database schema is the content of the database at a given time point. We need a type *ID* of object identifiers to uniquely and efficiently identify the objects and to model objects in different classes and references to other objects. We use *db* as a name of the instance of the database schema.

Example 4.4. An instance of the above university database schema is presented as following:

$db(\text{PERSONC}) =$

$\{(i_{101}, (\text{PersonID} : 1001, (\text{FName: John,LName: Dever,\{Professor,Dr\}}),$

Address: 28 Victoria Av, DoB:16/Jan/1953)),

$(i_{102}, (\text{PersonID} : 1002, (\text{FName: Allan, LName: Barry, Title: \{Senior Lecturer, Dr\}}),$

Address: 66 Albert St, DoB:23/Feb/1958)),

$(i_{103}, (\text{PersonID} : 2010, (\text{FName: Shirley, Churchill,Title: \{Lecturer\}}),$

Address: 531 Tramine Av., DoB:10/Oct/1960)),

$(i_{104}, (\text{PersonID} : 3203, (\text{FName: Jerry, LName:Hubbard,Title: \{HoD, Professor, Dr\}}),$

Address: 32 Ada St, DoB:02/May/1945)),

$(i_{105}, (\text{PersonID} : 2618, (\text{FName: LName: James, LName: Hooks,Title: \{Lecturer\}}),$

Address:116 College St, DoB:28/Jun/1960)),

$(i_{106}, (\text{PersonID} : 4322, (\text{FName: LName: Jill, Heslop,Title: \{ \}}),$

Address: 22 Fegerson St, DoB:11/Jul/1985)),

$(i_{107}, (\text{PersonID} : 4198, (\text{FName: LName: Frances,LName: Caban,Title: \{ \}}),$

Address: 78 Cuba St, DoB:30/Nov/1983)),

$(i_{108}, (\text{PersonID} : 4077, (\text{FName: LName: Lindsay,LName:Hamilton,Title: \{ \}}),$

Address: 29 Church St, DoB:06/Dec/1982)),

$(i_{109}, (\text{PersonID} : 4198, (\text{FName: LName: Jeff, LName: Perera,Title: \{ \}}),$

Address: 99 Broadway Av, DoB:18/Aug/1980)),

$(i_{110}, (\text{PersonID} : 2396, (\text{FName: Lindsay, LName: Kirton,Title: \{ \}}),$

Address:195 King St, DoB:03/Sept/1975))$\}$

$db(\text{SEMESTERC}) =$

$\{(i_{201}, (\text{Year} : 2000, \text{Season} : 01)),$

$(i_{202}, (\text{Year} : 2000, \text{Season} : 02)),$

$(i_{203}, (\text{Year} : 2001, \text{Season} : 01)),$

$(i_{204}, (\text{Year} : 2001, \text{Season} : 02)),$

$(i_{205}, (\text{Year} : 2002, \text{Season} : 01)),$

$(i_{206}, (\text{Year} : 2002, \text{Season} : 02)),$

$(i_{207}, (\text{Year} : 2003, \text{Season} : 01))\}$

$db(\text{ROOMC}) =$

$\{(i_{301}, (\text{Building} : \text{SSLB}, \text{NO} : 2, \text{Campus} : \text{PN})),$

$(i_{302}, (\text{Building} : \text{SST}, \text{NO} : 1, \text{Campus} : \text{WN})),$

$(i_{303}, (\text{Building} : \text{Marsdon}, \text{NO} : 1, \text{Campus} : \text{PN})),$

$(i_{304}, (\text{Building} : \text{AH}, \text{NO} : 2, \text{Campus} : \text{ALB})),$

$(i_{305}, (\text{Building} : \text{BSC}, \text{NO} : 203, \text{Campus} : \text{ALB}))\}$

$db(\text{DEPARTMENTC}) =$

$\{(i_{401}, (\text{Dname: Information Systems}, \{(\text{TelNumber} : 063566199, \text{Campus:PN}),$

$(\text{TelNumber} : 045763112, \text{Campus:WN})\})),$

$(i_{402}, (\text{Dname: Accounting}, \{(\text{TelNumber} : 063563188, \text{Campus:PN}),$

$(\text{TelNumber} : 098132699, \text{Campus:ALB})\})),$

$(i_{403}, (\text{Dname: Marketing}, \{(\text{TelNumber} : 063564132, \text{Campus:PN}),$

$(\text{TelNumber} : 045663188, \text{Campus:WN})\}))\}$

$db(\text{COURSEC}) =$

$\{(i_{501}, (\text{CNumber} : 110.001, \text{CName:Accounting Principle}, \text{Requires: } \{\})),$

$(i_{502}, (\text{CNumber} : 110.105, \text{Taxation}, \text{Requires: } \{i_{501}\})),$

$(i_{503}, (\text{CNumber} : 156.100, \text{Principle of Marketing}, \text{Requires: } \{\ \}))$,

$(i_{504}, (\text{CNumber} : 157.221, \text{Information Systems Analysis}, \text{Requires: } \{i_{503}\ \}))$,

$(i_{505}, (\text{CNumber} : 157.331, \text{Database Concepts}, \text{Requires: } \{i_{504}\ \}))\}$

$db(\text{LECTURERC}) =$

$\{(i_{101}, (\text{PersonID} : 1001, (\text{FName: John, LName:Dever, Title: \{Professor, Dr\}}),$

Specialization: Commercial Law, Department:i_{i402}, Campus:ALB)),

$(i_{102}, (\text{PersonID} : 1002, (\text{FName: Allan, LName:Barry, Title: \{Senior Lecturer, Dr\}}),$

Specialization: Pricing, Department:i_{i403}, Campus:PN)),

$(i_{103}, (\text{PersonID} : 2010, (\text{FName: Shirley, LName:Churchill, Title: \{Lecturer\}}),$

Specialization: Databases, Department:i_{i401}, Campus:WN)),

$(i_{104}, (\text{PersonID} : 1002, (\text{FName: Jerry, LName:Hubbard, Title: \{HoD Professor, Dr\}}),$

Specialization: Distributed Systems, Department:i_{i401}, Campus:PN)),

$(i_{105}, (\text{PersonID} : 1002, (\text{FName: James, LName:Hooks, Title: \{Lecturer\}}),$

Specialization: Culture and Accounting, Department:i_{i402}, Campus:WN))\}

$db(\text{PROJECTC}) =$

$\{(i_{701}, (\text{PNumber: p0001, PName: DIMO, Begin: Jun 2000,}$

End: Dec 2002, Description: Multilevel transaction...,

Budget: 100000,PrimaryInvestigator : $\{i_{103}, i_{104}\})$),

$(i_{702}, (\text{PNumber: p0201,PName: Consumer Behavior, Begin: Jan 2002,}$

End: Dec 2002, Description: The patterns of...,

Budget: 3000,PrimaryInvestigator : $\{i_{102}\})$),

$(i_{703}, (\text{PNumber: p0301,PName: Small Business Accounting, Begin: Feb 2003,}$

End: Dec 2003, Description: Small businesses are...,

Budget: 5000,PrimaryInvestigator : $\{i_{101}, i_{110}\}))\}$

$db(\text{PAPERC}) =$

$\{(i_{801}, (\text{Course: } : i_{501}, \text{Semester:}i_{201}, \text{Campus:ALB, Lecturer:}i_{101},$

$\{(\text{Day:Mon, Hour: 10am, Room:}i_{304}),$

$(\text{Day:Thur, Hour: 2pm, Room:}i_{304})\})),$

$(i_{802}, (\text{Course: } : i_{503}, \text{Semester:}i_{202}, \text{Campus:ALB, Lecturer:}i_{102},$

$\{(\text{Day:Wed, Hour: 9am, Room:}i_{305}),$

$(\text{Day:Fri, Hour: 1pm, Room:}i_{305})\})),$

$(i_{803}, (\text{Course: } : i_{502}, \text{Semester:}i_{203}, \text{Campus:WN, Lecturer:}i_{105},$

$\{(\text{Day:Tues, Hour: 1pm, Room:}i_{302}),$

$(\text{Day:Thur, Hour: 3pm, Room:}i_{302})\})),$

$(i_{804}, (\text{Course: } : i_{504}, \text{Semester:}i_{204}, \text{Campus:WN, Lecturer:}i_{103},$

$\{(\text{Day:Mon, Hour: 9am, Room:}i_{302}),$

$(\text{Day:Wed, Hour: 1pm, Room:}i_{302})\}))\}$

$(i_{805}, (\text{Course: } : i_{504}, \text{Semester:}i_{205}, \text{Campus:PN, Lecturer:}i_{104},$

$\{(\text{Day:Mon, Hour: 11am, Room:}i_{301}),$

$(\text{Day:Wed, Hour: 2pm, Room:}i_{301})\})),$

$(i_{806}, (\text{Course: } : i_{505}, \text{Semester:}i_{206}, \text{Campus:PN, Lecturer:}i_{104},$

$\{(\text{Day:Mon, Hour: 11am, Room:}i_{301}),$

$(\text{Day:Wed, Hour: 2pm, Room:}i_{303})\})),$

$(i_{807}, (\text{Course: } : i_{505}, \text{Semester:}i_{207}, \text{Campus:PN, Lecturer:}i_{104},$

$\{(\text{Day:Tues, Hour: 10am, Room:}i_{301}),$

$(\text{Day:Thur, Hour: 3pm, Room:}i_{303})\}))\}$

$db(\text{STUDENTC}) =$

$\{(i_{106}, (\text{PersonID:4322, StudentID: 99368, Supervisor : } i_{101},$

$\text{Major: } i_{402}, \text{Minor: } i_{403}, \{(\text{Paper } : i_{801}, \text{Grade: } A^{+}),$

(Paper: i_{802}, Grade: B^+)})),

(i_{107}, (PersonID:4198, StudentID: 00695, Supervisor : i_{105},

Major: i_{402}, Minor: i_{401}, {(Paper : i_{803}, Grade: B),

(Paper: i_{804}, Grade: A)})),

(i_{108}, (PersonID:4077, StudentID: 01396, Supervisor : i_{105},

Major: i_{401}, Minor: NUL, {(Paper : i_{805}, Grade: A^-),

(Paper: i_{807}, Grade:A)}))

(i_{109}, (PersonID:4198, StudentID: 02396, Supervisor : i_{105},

Major: i_{401}, Minor: NUL, {(Paper : i_{805}, Grade: B^+),

(Paper: i_{807}, Grade: NUL)}))}

□

4.3 Queries

For object oriented databases we must define some basic query algebra that can be used to define queries on database. To define such a query algebra we have to refer to elements of classes and their components. For this I will define *path expressions*. The discussion in the following subsection will cover the situations of path expressions defined on elements of base types, record types, set types, union types and references.

4.3.1 Path Expressions

In the relational data model, only the record type constructor is used and each attribute is defined on a base type. In the case of the object oriented data model, the whole underlying type system includes not only record type constructors but also some other bulk type constructors. A type system can be expressed as [28, p. 9]:

$$t \quad = \quad b \mid x \mid (a_1 : t_1, ..., a_n : t_n) \mid \{t\} \mid (a_1 : t_1) \cup \cdots \cup (a_n : t_n)$$

In fact, there could be further type constructors, e.g. for lists and multisets, but I will concentrate on this simplified type system here. Details of the above type system have been provided in Section 4.1. In object oriented databases, for a given class C with structure expression exp_C and representation type T_C, a database instance of a class C satisfies

$$db(C) \subseteq \{(i, v) \mid i \in dom(ID), v \in dom(T_C)\}$$

The complete definition of a database (or instance of a database schema) was given in Definition 4.5. We define the path expressions as below.

Definition 4.6. Let C be a class with structure expression exp_C and representation type T_C. *Path expressions* for class C (or exp_C) may have the following formats:

(i) *ident* of type ID,

(ii) *value* of type T_C,

(iii) if exp_C uses a record type, i.e. $exp_C = (a_1 : exp_1, \ldots, a_n : exp_n)$, then we get path expressions:

- $value.a_i$ of type T_i which is a representative type for exp_i and $1 \leq i \leq n$,
- $value!a_i$ of type T_D if a_i is a reference to class D, i.e. $a_i : exp_i = a_i : D$,
- $value.a_i.path_i$ where $path_i$ is a path expression for exp_i,
- $value!a_i.path_i$ where a_i is a reference to class D, i.e. $a_i : exp_i = a_i : D$ and $path_i$ is a path expression for exp_D.

(iv) if exp_C use a union type, i.e. $exp_C = (a_1 : exp_1) \cup \cdots \cup (a_n : exp_n)$, then we get path expression:

- $value.a_i$ of type T_i which is the representation type for exp_i,
- $value.a_i.path_i$ where $path_i$ is a path expression for exp_i.

(v) if exp_C uses a set type, i.e. $exp_C = \{exp\}$, then we obtain path expression:

- *value* of type $T_C = \{exp\}$,
- *value.path* where *path* is a path expression corresponding to *exp*.

(vi) if exp_C uses only a reference , i.e. $exp_C = r : D$, then we obtain path expression: $value!r.path$ where *path* is path expression for the class D.

Remark: If $r : D$ appears in exp_C, i.e. $exp_C = \ldots, a_i : r : D, \ldots$, we get $T_i = ID$. Thus path expressions are:

- $value!a_i.r$ of type T_D,
- If T_D has structure expression exp_D, eg. $exp_D = (a_{i1} : exp_{i1}, \ldots, a_{in} : exp_{in})$. Then this situation is treated as if $r : D$ is replaced by exp_D. Therefore the structure expression of class C can be represented with $exp_C = \ldots, a_i : exp_D, \ldots$. The representation type of class C is $T_C = \ldots a_i : T_D \ldots$. Then we get path expression

$$value!a_i$$

to refer to the value of type T_D, i.e. if $exp_D = (a_{i_1} : exp_{i_1}, \ldots, a_{i_n} : exp_{i_n})$, We can use

$$value!a_i.a_{i_j}$$

to refer to the value of the component in exp_D and handle it as if there were no references.

If *path* is a path expression, we use T_{path} to denote the representation type of the structure expression that is identified by the *path*.

Example 4.5. We choose class LECTURERC from university database schema. `Class` LECTURERC IsA PERSONC `Struct` (PersonID: *NAT*, $\underbrace{\text{Name}}_{path_1}$: ($\underbrace{\text{FName}}_{path_2}$: *STRING*,LName: *STRING*, $\underbrace{\text{Title}}_{path_3}$: {*STRING*}), $\underbrace{\text{Specialization}}_{path_4}$: *STRING*, $\underbrace{\text{Department}}_{path_5}$: DEPARTMENTC, Campus: *STRING*)

There are some path expressions defined on class LECTURERC as following:

(i) *ident*

(ii) *value*

(iii) $path_1 = value.\text{Name}$

$path_2 = value.\text{Name.FName}$

$path_3 = value.\text{Title}$

$path_4 = value.\text{Specialization}$

(iv) $path_5 = value!\text{Department}$

$path'_5 = value!\text{Department.Name}$

$path_5$ refers to the identifier of the department being referenced and $path'_5$ refers to the name of the referenced department.

□

4.3.2 Queries

From application information we can get a set of queries accessing databases. For the relational model, we can use relational algebra to model queries and to optimize queries. We must provide a model for querying object oriented databases. Basically, an algebra in the general mathematical sense is given by a set $\mathcal{A}$ and a set of operations *op* on $\mathcal{A}$ [29]. In the object oriented model, each query results in a set of pairs (id, v), where id is an identifier and v is a value of some proper type. The value v may contain identifiers, which must appear in the database, to which the query is applied. More generally, it would also be possible that the identifiers appear in the query result, but much more complicated queries are not handled here. We refer to such a set of pairs as a "class instance". A query Q on $\mathcal{S}$ consists of a structure expression exp_Q called answer schema (with all references pointing to classes in $\mathcal{S}$) and an algebra expression q, i.e. a query Q is defined in the form $Q = (exp_Q, q)$. Every operator in the query algebra accepts (one or two) class instances as arguments and returns another class instance as a result [25].

We start by saying that every class name $C \in \mathcal{S}$ can be considered as a query $q = C$. The answer schema is exp_C itself and database instance db is mapped via query q to $db(Q)$, which is a new class instance $db(Q)$ extending db over $\mathcal{S}$ to a new database, which contains a set of new objects with new identifiers created for each of them. Also, pairs $(v : T)$ with value v of type T is considered as a query.

Definition 4.7. Let $\mathcal{S}$ be a database schema, db be a database instance over $\mathcal{S}$, $id(db)$ be the set of all identifiers appearing in db, $db(Q)$ be the resulting class instance of evaluating query algebra $Q = (exp_Q, q)$. There are two basic query algebra operations:

(i) $q = C$ with C is a class name appeared in $\mathcal{S}$ with a query expressed as $Q = (exp_C, q)$, resulting in $db(Q) = \{(id_w, v) \mid id_w :: ID, id_w \notin id(db).\exists id :: ID.(id, v) \in db(C)\}$.

(ii) $q = (v : T)$ with a type T and a value v of type T and $Q = (T, q)$, resulting in

$db(Q) = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db)\}$.

Example 4.6. the university database instance db in Example 4.4 contains an instance $db(\text{RoomC})$ of class RoomC:

$$\begin{aligned} db(\text{RoomC}) = \{&(i_{301}, (\text{Building} : \text{SSLB}, \text{NO} : 2, \text{Campus} : \text{PN})), \\ &(i_{302}, (\text{Building} : \text{SST}, \text{NO} : 1, \text{Campus} : \text{WN})), \\ &(i_{303}, (\text{Building} : \text{Marsdon}, \text{NO} : 1, \text{Campus} : \text{PN})), \\ &(i_{304}, (\text{Building} : \text{AH}, \text{NO} : 2, \text{Campus} : \text{ALB})), \\ &(i_{305}, (\text{Building} : \text{BSC}, \text{NO} : 203, \text{Campus} : \text{ALB}))\} \end{aligned}$$

A query operation q =RoomC for a query $Q_1 = (exp_{\text{RoomC}}, \text{RoomC})$ with a resulting

instance of Q as

$$\begin{aligned} db(Q_1) = \{&(i_{1001}, (\text{Building} : \text{SSLB}, \text{NO} : 2, \text{Campus} : \text{PN})), \\ &(i_{1002}, (\text{Building} : \text{SST}, \text{NO} : 1, \text{Campus} : \text{WN})), \\ &(i_{1003}, (\text{Building} : \text{Marsdon}, \text{NO} : 1, \text{Campus} : \text{PN})), \\ &(i_{1004}, (\text{Building} : \text{AH}, \text{NO} : 2, \text{Campus} : \text{ALB})), \\ &(i_{1005}, (\text{Building} : \text{BSC}, \text{NO} : 203, \text{Campus} : \text{ALB}))\} \end{aligned}$$

Note the result of the query on $db(C)$ is a new class that contains a set of objects with new identifiers which have not appeared in the database. But the values for all the attributes are the same.

Another query Q_2 =((Building: *STRING*, NO: *NAT*, Campus: *STRING*),(Building: SSLB, NO: 2, Campus: PN)) results:

$db(Q_2) = \{(i_{98001}, (\text{Building} : \text{SSLB}, \text{NO} : 2, \text{Campus} : \text{PN}))\}$ □

We can define a selection operation with a query algebra q. To do this we need to use path expressions *path* for class C to define selection formulae. Path expressions have been defined in the previous subsection.

Definition 4.8. Let C be a class, *path* be path expressions on class C, take either

- two path expression $path_1, path_2$ of C of some type,
- or a path expression *path* on class C and a value v of the type of *path*.

Selection formulae φ can be defined in either of the following forms:

- $\varphi \equiv path_1 = path_2$,
- or $\varphi \equiv path = v$.

Definition 4.9. (selection)

Let $\mathcal{S}$ be a database schema, db be a database instance over $\mathcal{S}$, $id(db)$ be the set of all identifiers appearing in db. Take any query $Q = (exp_Q, q)$ and a selection formula φ for exp_Q we get:

- a query algebra $q' = \sigma_\varphi(Q)$ for selection operation,
- a query $Q' = (exp_Q, q')$ with $exp_{Q'} = exp_Q$,
- evaluating Q' on db results in a database instance:

  $db(Q') = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id, v) \in db(Q) \wedge \varphi(v) = true\}$

Example 4.7. There is a database instance db over the university database schema. With a query algebra $q = \text{PERSONC}$ we have a query $Q = (exp_{\text{PERSONC}}, q)$, then db is extended by adding a $db(Q)$ within which all the objects' identifiers have not occurred before.

Then we have a selection formula: $\varphi \equiv value.\text{Name.FName} = \text{'John'}$. Using query operation $\sigma_\varphi(Q)$ we get a result as

$db(Q') = \{(i_{96101}, (\text{PersonID} : 1001, (\text{FName: John,LName: Dever,\{Professor,Dr\}}),$

Address: 28 Victoria Av, DoB:16/Jan/1953))} □

Definition 4.10. (renaming operation)

Let $Q = (exp_Q, q)$ be a query, $attr(Q)$ denote all the attribute names appearing in exp_Q with $a_i \in attr(Q)$. A rename operation renames some of the attributes of a class. We have following expressions:

- query algebra $q' = \rho_{a_1 \rightarrowtail b_1, \ldots, a_n \rightarrowtail b_n}(Q)$,
- new query $Q' = (exp'_Q, q')$,
- Evaluating query Q' on db we get:

  $db(Q') = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id, v) \in db(Q)\}$

with b_i as a new name for attribute a_i.

In order to define generalized projection, we need to use the definition of super type that is introduced in [26]. We use the form $T_{C_2} \leq T_{C_1}$ to express that T_{C_1} is a super type of type T_{C_2}. This expression indicates a mapping:

$$\pi_{C_1}^{C_2} : dom(T_{C_2}) \rightarrow dom(T_{C_1})$$

We can get the following definition for generalized projection.

Definition 4.11. (generalized projection)

Let $Q = (exp_Q, q)$ be a query, $exp_{Q'}$ be a new structure expression which is a super structure expression of exp_Q, i.e. $T_Q \leq T_{Q'}$ for the representation types. A generalized projection is a mapping:

$$\pi_{Q'}^{Q} : dom(T_Q) \rightarrow dom(T_{Q'})$$

results in a new query: $Q' = (exp_{Q'}, \pi_{Q'}(Q))$,

with $db(Q') = \{(id_w, v') \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id, v) \in db(Q).v' = \pi_{Q'}^{Q}(v)\}$.

Definition 4.12. (join)

Let $Q_1 = (exp_{Q_1}, q_1)$, $Q_2 = (exp_{Q_2}, q_2)$ be two queries, exp be a common super structure expression with $exp_{Q_i} \leq exp$ $(i = 1, 2)$, then there exists:

- a new structure for join operation $exp_1 \bowtie_{exp} exp_2$,
- with new query $Q_1 \bowtie_{exp} Q_2 = (exp_{Q_1} \bowtie_{exp} exp_{Q_2}, q_1 \bowtie_{exp} q_2)$,
- and a new instance

$$\begin{aligned} db(Q_1 \bowtie_{exp} Q_2) = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id_1, v_1) \in db(Q_1). \\ (id_2, v_2) \in db(Q_2).\pi_{exp}^{Q_1}(v_1) = \pi_{exp}^{Q_2}(v_2) \wedge \pi_{Q_1}^{Q_1 \bowtie Q_2}(v) = v_1 \wedge \\ \pi_{Q_2}^{Q_1 \bowtie Q_2}(v) = v_2\} \end{aligned}$$

Definition 4.13. (set operations)

Let $Q_1 = (exp_Q, q_1)$, $Q_2 = (exp_Q, q_2)$ be two given queries.

- union operation on sets is expressed with structure expression:

$$exp_1 \cup exp_2$$

with a new query $Q_1 \cup Q_2 = (exp_Q, q_1 \cup q_2)$. Evaluating q we get

$$db(Q_1 \cup Q_2) = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id_1, v) \in db(Q_1) \\ \vee (id_2, v) \in db(Q_2)\}$$

- intersection of two sets is expressed with structure expression:

$$exp_1 \cap exp_2$$

with a new query $Q_1 \cap Q_2 = (exp_Q, q_1 \cap q_2)$. Evaluating q we get

$$db(Q_1 \cap Q_2) = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id_1, v) \in db(Q_1) \\ \wedge (id_2, v) \in db(Q_2)\}$$

- difference operation on sets is expressed with structure expression:

$$exp_1 - exp_2$$

with a new query $Q_1 \cup Q_2 = (exp_Q, q_1 - q_2)$. Evaluating q we get

$$db(Q_1 - Q_2) = \{(id_w, v) \mid id_w :: ID \wedge id_w \notin id(db).\exists id :: ID.(id_1, v) \in db(Q_1) \\ \wedge (id_1, v) \notin db(Q_2)\}$$

With all the definitions discussed above we can define queries that access databases. In particular, we can define simple predicates for horizontal fragmentation in object oriented databases.

Chapter 5

Fragmentation Operations in Object Oriented Databases

Distribution design involves making decisions on the fragmentation and placement of data across the sites of a computer network. The design process has two phases: fragmentation and allocation. Fragmentation of object oriented database systems is a complex problem because:

- it involves set-valued and reference attributes,
- inheritance (IsA) relationships are employed,
- complex data types should be considered.

In the object oriented environment, fragmenting a class may use three techniques: horizontal and vertical fragmentation as well as splitting techniques. It is also possible to combine all these techniques to perform mixed (hybrid) fragmentation of a class.

Horizontal fragmentation exploits databases to be defined by sets. It partitions a class into a set of new classes (fragments), which will have exactly the same structure but different contents. Thus, a horizontal fragment of a class contains a subset of the whole class instance.

On the other hand, *vertical fragmentation* exploits the tuple type constructor. Vertical fragmentation results in fragments with different new structures. There may be more general approaches to vertical fragmentation. There are some interesting problems to be solved when generalizing vertical fragmentation from the RDM to the OODM.

The third operation that is available in the object oriented database environment is *splitting* [28]. The splitting operation will split classes and introduce new references. The results of the splitting operation is to replace a class with two new ones without changing the information.

In the sections below, the above three techniques will be reviewed and defined. Some examples will be given to explain each of them.

5.1 Split Fragmentation

Split fragmentation is first introduced in [31] as a database design primitive. This operation results in one class being replaced with two classes, one referencing the other.

Definition 5.1. For a given class $C \in \mathcal{S}$ with structure expression exp_C, if a structure expression exp occurs within the structure exp_C, then *split fragmentation* results in a new class C' to be added to the schema $\mathcal{S}$ such that:

(i) $exp_{C'} = exp$.

(ii) exp in exp_C to be replaced by a new reference $r' : C'$.

Example 5.1. In the university schema $\mathcal{S}$, there is a class:

`Class` PAPERC `Structure` (Course: COURSEC,Semester: SEMESTERC, Campus: *STRING*, Lecturer: LECTURERC, {(Day: *DATE*, Hour: *NAT*,Room: ROOMC)})

From the above schema we note that class PAPERC contains the structure expression 'Campus: *STRING*'. That means $exp = STRING$ occurs within the structure exp_{PAPER}. By adding a new class SITEC in the schema $\mathcal{S}$ with $exp_{SITE} = STRING$, the structure expression $exp = STRING$ can be replaced by SITEC.

Therefore schema $\mathcal{S}$ should be redefined as following: `Class` PAPERC `Structure` (Course: COURSEC,Semester: SEMESTERC, Campus: SITEC, Lecturer: LECTURERC, {(Day: *DATE*, Hour: *NAT*,Room: ROOMC)})

`Class` SITEC `Struct` *STRING*.

Accordingly, the database instances of class PAPERC presented in example 4.4 will be changed to:

$db(\text{PAPERC}) =$

$\{(i_{1801}, (\text{Course:}i_{501}, \text{Semester:}i_{201}, \text{Campus:}i_{903}, \text{ Lecturer:}i_{101},$

$\{(\text{Day:Mon, Hour: 10am, Room:}i_{304}), (\text{Day:Thur, Hour: 2pm,Room:}i_{304})\})),$

$(i_{1802}, (\text{Course: : }i_{503}, \text{Semester:}i_{202}, \text{Campus: }i_{903}, \text{ Lecturer:}i_{102},$

$\{(\text{Day:Wed, Hour: 9am, Room:}i_{305}), (\text{Day:Fri, Hour: 1pm,Room:}i_{305})\})),$

$(i_{1803}, (\text{Course: : }i_{502}, \text{Semester:}i_{203}, \text{Campus:}i_{902}, \text{ Lecturer:}i_{105},$

$\{(\text{Day:Tues, Hour: 1pm, Room:}i_{302}), (\text{Day:Thur, Hour: 3pm,Room:}i_{302})\})),$

$(i_{1804}, (\text{Course: : }i_{504}, \text{Semester:}i_{204}, \text{Campus:}i_{902}, \text{ Lecturer:}i_{103},$

$\{(\text{Day:Mon, Hour: 9am, Room:}i_{302}), (\text{Day:Wed, Hour:1pm,Room:}i_{302})\})\}$

$(i_{1805}, (\text{Course: : }i_{504}, \text{Semester:}i_{205}, \text{Campus:}i_{901}, \text{ Lecturer:}i_{104},$

$\{(\text{Day:Mon, Hour: 11am, Room:}i_{301}), (\text{Day:Wed, Hour: 2pm,Room:}i_{301})\})),$

$(i_{1806}, (\text{Course: : }i_{505}, \text{Semester:}i_{206}, \text{Campus:}i_{901}, \text{ Lecturer:}i_{104},$

$\{(\text{Day:Mon, Hour: 11am, Room:}i_{301}), (\text{Day:Wed, Hour: 2pm,Room:}i_{303})\})),$

$(i_{1807}, (\text{Course: : }i_{505}, \text{Semester:}i_{207}, \text{Campus:}i_{901}, \text{ Lecturer:}i_{104},$

$\{(\text{Day:Tues, Hour:10am,Room:}i_{301}), (\text{Day:Thur, Hour: 3pm,Room:}i_{303})\})\}$

$db(\text{SITE}) =$

$\{(i_{901}, \text{PN}), (i_{902}, \text{WN}), (i_{903}, \text{ALB})\}$

□

5.2 Horizontal Fragmentation

5.2.1 Horizontal Fragmentation on Class Level

Schewe [28] makes a first effort to generate horizontal fragmentation techniques based on the object oriented database model introduced in [30].

Definition 5.2. Let C be some classes, φ_i be a Boolean valued function, $db(C)$ be an instance of class C. Horizontal fragmentation of class C replaces C by new classes $C_1, \ldots, C_n$ such that:

(i) $exp_{C_i} = exp_C$

(ii) each instance $db(C)$ of class C can be split into pairwise disjoint database instances $db(C_1), \ldots, db(C_n)$ over $\{C_1, \ldots, C_n\}$ such that :

$$db(C) = \bigcup_{i=1}^{n} db(C_i) \quad \text{with disjoint sets} \quad db(C_i)$$

(iii) if there is a class D referencing C, i.e. $r : C$ occurs in exp_D, the references have to be replaced as well, i.e. $r : C$ be replaced by $(a_1 : r_1 : C_1), \ldots, (a_n : r_n : C_n)$ with new pairwise distinct reference names $r_1, \ldots, r_n$

By using the algebra introduced in section 4.3, the horizontal fragmentation on class level can be expressed as:

$$db(C_i) \quad = \quad \sigma_{\varphi_i}(db(C)), \quad 1 \leq i \leq n, \tag{5.1}$$

where φ_i is the selection formula used to obtain fragment C_i. In Chapter 6, we will discuss 'normal predicates' for horizontal fragmentation. These will generalize the minterm predicates used for the relational data model as discussed in Chapter 3.

Example 5.2. Again we take the university schema and horizontal fragment class LECTURERC by using

$$\varphi_1 \equiv \text{Campus} = \text{'PN'} \quad \varphi_1 \equiv \text{Campus} = \text{'WN'} \quad \varphi_1 \equiv \text{Campus} = \text{'ALB'}$$

Then class LECTURERC will be partitioned into three new classes with the following structures:

`Class` LECTURERC$_1$ IsA PERSONC `Structure` (PersonID: *NAT*, Name: *PERSONNAME*, Specialization: *STRING*, Department: DEPARTMENTC, Campus: *STRING*)

`Class` LECTURERC$_2$ IsA PERSONC `Structure` (PersonID: *NAT*, Name: *PERSONNAME*, Specialization: *STRING*, Department: DEPARTMENTC, Campus: *STRING*)

`Class` LECTURERC$_3$ IsA PERSONC `Structure` (PersonID: *NAT*, Name: *PERSONNAME*, Specialization: *STRING*, Department: DEPARTMENTC, Campus: *STRING*)

`Class` PAPERC `Structure` (Lecturer: (h$_1$:LECTURERC$_1$ $\cup$ h$_2$:LECTURERC$_2$ $\cup$ h$_3$:LECTURERC$_3$), Semester: SEMESTERC, Campus: *STRING*, Course: COURSEC, {(Room: ROOMC, Hour: *NAT*, Day: *DATE*)})

The database instances of class PAPERC and class LECTURERC from example 4.4 are then fragmented into:

db(LECTURERC$_1$) =

(i_{2102}, (PersonID:1002, (FName: Allan,LName:Barry, Title: {Senior Lecturer, Dr}), Specialization: Pricing,Department:i_{i403}, Campus:PN)),

(i_{2104}, (PersonID:1002, (FName: Jerry,LName:Hubbard, Title: {HoD Professor, Dr}), Specialization: Distributed Systems,Department:i_{i401}, Campus:PN)),

(i_{2105}, (PersonID:1002, (FName: James,LName:Hooks, Title: {Lecturer}), Specialization: Culture and Accounting, Department:i_{i402}, Campus:PN))}

db(LECTURERC$_2$) =

(i_{2103}, (PersonID:2010, (FName: Shirley,LName:Churchill, Title: {Lecturer}), Specialization: Databases,Department:i_{i401}, Campus:WN))

db(LECTURERC$_3$) =

{(i_{2101}, (PersonID:1001, (FName: John, LName:Dever, Title: {Professor, Dr}),

Specialization: Commercial Law,Department:i_{i402}, Campus:ALB))

$db(\text{PAPERC}) =$

$\{(i_{801}(\text{Course:}i_{501}, \text{Semester:}i_{201}, \text{Campus:ALB, Lecturer: } \text{h}_3 : i_{101},$

$\{(\text{Day:Mon, Hour: 10am, Room:}i_{304}), (\text{Day:Thur, Hour: 2pm,Room:}i_{304})\})),$

$(i_{802}(\text{Course: : } i_{503}, \text{Semester:}i_{202}, \text{Campus: ALB, Lecturer: } \text{h}_1 : i_{102},$

$\{(\text{Day:Wed, Hour: 9am, Room:}i_{305}), (\text{Day:Fri, Hour: 1pm,Room:}i_{305})\})),$

$(i_{803}(\text{Course: : } i_{502}, \text{Semester:}i_{203}, \text{Campus:WN, Lecturer: } \text{h}_1 : i_{105},$

$\{(\text{Day:Tues, Hour: 1pm, Room:}i_{302}), (\text{Day:Thur, Hour: 3pm,Room:}i_{302})\})),$

$(i_{804}(\text{Course: : } i_{504}, \text{Semester:}i_{204}, \text{Campus:WN, Lecturer: } \text{h}_2 : i_{103},$

$\{(\text{Day:Mon, Hour: 9am, Room:}i_{302}), (\text{Day:Wed, Hour:1pm,Room:}i_{302})\}))$

$(i_{805}(\text{Course: : } i_{504}, \text{Semester:}i_{205}, \text{Campus:PN, Lecturer: } \text{h}_1 : i_{104},$

$\{(\text{Day:Mon, Hour: 11am, Room:}i_{301}), (\text{Day:Wed, Hour: 2pm,Room:}i_{301})\})),$

$(i_{806}(\text{Course: : } i_{505}, \text{Semester:}i_{206}, \text{Campus:PN, Lecturer: } \text{h}_1 : i_{104},$

$\{(\text{Day:Mon, Hour: 11am, Room:}i_{301}), (\text{Day:Wed, Hour: 2pm,Room:}i_{303})\})),$

$(i_{807}(\text{Course: : } i_{505}, \text{Semester:}i_{207}, \text{Campus:PN, Lecturer: } \text{h}_1 : i_{104},$

$\{(\text{Day:Tues, Hour:10am,Room:}i_{301}), (\text{Day:Thur, Hour: 3pm,Room:}i_{303})\}))\}$

□

5.2.2 Horizontal Fragmentation on Type Level

Because the chosen underlying type system allows arbitrary nesting of type constructors, the set type constructor may appear within a structure expression, say exp_C.

Definition 5.3. For a given class C with structure expression exp_C, if a structure expression $\{exp\}$ appears in exp_C, horizontal fragmentation on type level can be performed with the following steps:

(i) apply the splitting operation which results in exp in T_C being changed to $r' : C'$ with new reference name r' and new class name C' and $T_{C'} = exp$.

(ii) apply horizontal fragmentation to C' by using selection operation:

$$db(C') = \bigcup_{i=1}^{n} \sigma_{\varphi_i}(db(C')) \quad \text{with disjoint sets} \quad \sigma_{\varphi_i}(db(C'))$$

This should lead to n new classes $C_1, ..., C_n$ all with $T_{C_i} = exp$. Since in this case there is exactly one reference to C' we would replace $\{exp\}$ in T_C by $\{(\ell_1 : r_1 : C_1) \cup \ldots \cup (\ell_n : r_n : C_n)\}$ or equivalently by $(\ell_1 : \{r_1 : C_1\}, \ldots, \ell_n : \{r_n : C_n\})$.

(iii) undo the splitting, which has the same effect as if $\{exp\}$ in T_C would just have been replaced by $(\ell_1 : \{exp\}, \ldots, \ell_n : \{exp\})$.

Example 5.3. In the university database schema there is a Class COURSEC with structure:

`Class` COURSEC

`Structure` *COURSE* ∘

(Requires: {r: COURSEC})

`End` COURSEC

There is a structure expression {r: COURSEC} that occurs in exp_{COURSE}. We replace {r : COURSEC} with C' together with new reference name r', and $T_{C'} = r$: COURSEC. Then we horizontally fragment COURSEC which results in n new classes $C_1, \ldots, C_n$ with $T_{C_i} = r$: COURSEC. {r: COURSEC} will be replaced by $(\ell_1 : \{r_1 : C_1\}, \ldots, \ell_n : \{r_n : C_n\})$. Finally, we undo splitting. {r: COURSEC} will be replaced by $(\ell_1 : \{r : \text{COURSEC}\}, \ldots, \ell_n : \{r : \text{COURSEC}\})$. The result should be:

`Class` COURSEC

`Structure` *COURSE*∘

(Requires: $(\ell_1 : \{r : \text{COURSEC}\}) \cup, \ldots, \cup (\ell_n : \{r : \text{COURSEC}\})$)

`End` COURSEC □

It is noticed that fragmentation on type level does not create new classes. Thus, it is more a restructuring than a fragmentation operation. However, type level horizontal fragmentation may enable a subsequent vertical fragmentation.

5.3 Vertical Fragmentation for Object Oriented Databases

5.3.1 Vertical Fragmentation on Class Level

Let us first review the properties of value-identifiability for a class offered in [28].

Definition 5.4. *Value-identifiability* for a class C means that for each database db and each $(i, v) \in db(C)$ there are must be a query that would result in v and nothing else.

Weak value-identifiability would allow to reach (i, v) by following a sequence of references and subclass links starting from a value-identiable class.

Schewe [28] makes some generalization from the relational datamodel to the OODM. The definition of vertical fragmentation on the class level can be summarized with the following definition.

Definition 5.5. Let C be some class in a database schema $\mathcal{S}$. Assume that the outermost constructor in the structure expression exp_C was the record type constructor, say $exp_C = (a_1 : exp_1, ..., a_n : exp_n)$. $db(C)$ indicates a instance of class C. Vertical fragmentation on class C replaces C by a set of new classes $C_1, ..., C_k$ with $exp_{C_i} = (a_1^i : exp_{i_1}, ..., a_{n_i}^i : exp_{i_{n_i}})$ such that:

(i) the attributes will be distributed

$$\{a_1, ..., a_n\} = \bigcup_{i=1}^{k} \{a_1^i, ..., a_{n_i}^i\}$$

(ii) $db(C)$ will be split into database instances $db(C_i)$ such that

$$db(C) = (\ldots((db(C_1) \bowtie_{t_{1,2}} db(C_2)) \bowtie_{t_{1,2,3}} \ldots) \bowtie_{t_{1,\ldots,k}} db(C_k))$$

where by using the algebra defined in section 4.3 $db(C_i) = \pi_{X_i}(db(C))$ with $X_i = \{a_1^i, ..., a_{n_i}^i\}$, $t_{1,\ldots,k} = (a_{x_1^i} : t_{x_1^i}, ..., a_{x_{l_i}^i} : t_{x_{l_i}^i})$ with $(a_{x_1^i}, \ldots, a_{x_{l_i}^i}) = (X_1 \cup ... \cup X_{i-1}) \cap X_i$ and t_x is the representation type for the structure expression exp_x.

(iii) if there are references to the class C, i.e. $r : C$ in some exp_D, $r : C$ in each exp_D will be replaced by a new structure expression with an outermost record constructor and new references $(b_1 : r_1 : C_1, \ldots, b_k : r_k : C_k)$.

(iv) to preserve value-identifiability, there must be at least one of the new classes C_i, which is value-identifiable, the others can be just weakly value-identifiable as the new classes will use all the same object identifiers, i.e. $(i, v_1) \in db(C_1), ..., (i, v_k) \in db(C_k)$. Then, $r : C$ can be simply replaced by $r : C_i$ where C_i must be a value-identifiable superclass for all the other new classes. According to the definition above, the other classes would be weakly value-identifiable.

Example 5.4. We apply this technique to class CourseC in the university database schema.

`Class` CourseC `Structure` *COURSE* ∘ (Requires: {r: CourseC})

Will be replaced by:

`Class` CourseC `Structure` *COURSE*

`Class` Course_PrerequisC IsA CourseC `Structure` (Requires: {r: CourseC})

The database instance will be accordingly vertically fragmented into the following two

fragments:

$$db(\text{COURSEC}) = \{(i_{3501}(\text{CNumber} : 110.001, \text{CName:Accounting Principle}))$$
$$(i_{3502}(\text{CNumber} : 110.105, \text{CName:Taxation})),$$
$$(i_{3503}(\text{CNumber} : 156.100, \text{CName:Principle of Marketing})),$$
$$(i_{3504}(\text{CNumber} : 157.221, \text{CName:Information Systems Analysis})),$$
$$(i_{3505}(\text{CNumber} : 157.331, \text{CName:Database Concepts}))\}$$

$$db(\text{COURSE\_PREREQUISC}) = \{(i_{3501}\text{Requires: } \{\ \}),$$
$$(i_{3502}, \text{Requires: } \{i_{3501}\}),$$
$$(i_{3503}, \text{Requires: } \{\ \}),$$
$$(i_{3504}, \text{Requires: } \{i_{3503}\ \}),$$
$$(i_{3505}, \text{Requires: } \{i_{3504}\ \})\}$$

□

Note that the assumption that exp_C involves the record constructor as outermost type constructor is a restriction that may not be needed in general. However, without this assumption reconstruction may become very difficult. Therefore, I need not investigate this possibility of being more general here.

5.3.2 Vertical Fragmentation on Type Level

Vertical fragmentation can also be performed on type level if a tuple constructor occurs inside the nested structure of a class. The original database could be reconstructible by using a generalized join operation [26].

Definition 5.6. For a given class C with structure expression exp_C, if a tuple type constructor $(a_1 : exp_1, ..., a_n : exp_n)$ is used inside exp_C, *vertical fragmentation on type level* is performed with the following steps [28]:

(i) introduce a new class C' with $exp_{C'} = (a_1 : exp_1, ..., a_n : exp_n)$,

(ii) replace $(a_1 : exp_1, ..., a_n : exp_n)$ in exp_C by a new reference $r' : C'$,

(iii) vertically fragment the new class C' into $C_1, ..., C_k$ and replace $r' : C'$ in exp_C by a new structure expression $(b_1 : r_1 : C_1, ..., b_k : r_k : C_k)$,

(iv) undo the splitting, i.e. replace $(a_1 : exp_1, ..., a_n : exp_n)$ by the new structure expression $(b_1 : exp'_1, ..., b_k : exp'_k)$, where exp'_i is the structure expression of C_i that has the form $(a_{j_1^i} : exp_{j_1^i}, ..., a_{j_{m_i}^i} : exp_{j_{m_i}^i})$.

Same as horizontal fragmentation on type level, vertical fragmentation on type level is also merely restructuring in preparation for subsequent fragmentation steps. Therefore, it is not necessary to further discuss fragmentation on type level here.

5.4 Fragmentation Strategies

The fragmentation of object oriented databases can contain the following rules:

(i) perform splitting first as the result can be used to allow subsequent horizontal and vertical fragmentation of classes;

(ii) then apply horizontal and vertical fragmentation operation on classes;

(iii) adapt other classes which reference the classes that have been fragmented;

Though I do not discuss methods, a fourth step would be to adapt the methods and queries that access the database.

We should adapt the *correctness rules* of fragmentation in distributed relational databases to our object oriented environment to check whether the techniques proposed are correct. They are:

(i) *Completeness* requires that no information is lost.

(ii) *Disjointness* requires that no information will be duplicated.

(iii) *Reconstruction* requires that there is a unique way to reconstruct a non-fragmented database from its fragments.

The input to the design process is a global conceptual schema and access pattern information while the output of the process is a set of local conceptual schema [24]. By analyzing potential user requirements for both process and data, the input information can then be acquired.

5.5 Related Work

Some fragmentation techniques for distributed object oriented database systems have been found in the literature. For horizontal fragmentation, Ezeife and Barker [13] review a taxonomy of fragmentation problems in a distributed object base. They contribute a set of algorithms for horizontally fragmenting the four realizable class models on the taxonomy. These four class models include: simple attributes and simple methods, simple attributes and complex methods, complex attributes and simple methods, and complex attributes and complex methods. After some assumptions and definitions have been made, algorithms are presented. For the first model, a class is fragmented by applying an algorithm that has four steps. First they define the link graph of classes. Their second and third steps define the primary and derived horizontal fragmentation. Finally, the primary and derived fragments are combined according to two affinity rules.

Bellatreche, Kamalakar and Simonet [5] propose two horizontal fragmentation algorithms: the primary algorithm and the derived algorithm. For primary fragmentation, they study the role of queries and constructed a predicate affinity matrix with which frequencies of the queries was taken into account. The algorithm in Navathe & Karlapalem [21] is applied to form the predicate clusters in which the predicates have high affinity to one another. The set of predicates is optimized by using predicate implication. If there are predicates defined on some attributes or methods on which there are no predicates in a subset then it will be included in that subset to further modify the subset of predicates. Fragmentation will be

defined on each of the final modified predicate subsets. At last, a fragment will be defined by the negation of the disjunction of all predicates previously defined. A derived algorithm is defined with component predicates defined on a path. The advantage of this approach is that query frequencies have been considered when partitioning class and therefore decrease the number of fragments. But it creates overlap fragments that need an extra procedure to make them disjoint.

Chapter 6

A Method for Horizontal Fragmentation in Object Oriented Databases

The design of distributed object oriented databases (DOODBs) in principle follows the same procedure as for distribution design of relational databases. I will concentrate only on horizontal fragmentation here. According to this procedure reviewed in Section 3.6.1, the first step is to extract simple predicates from application information. The second step is the construction of minterm predicates and primary horizontal fragments according to the minterm predicates. The sections below will first define the format of simple predicates for object oriented databases. Then normal predicates will be introduced, which generalize the minterm predicates we used for the relational data model (RDM). Finally a general fragmentation process based on a cost model will be briefly defined.

6.1 Simple Predicates

Recall from Section 3.6.1 that the first step in the design of fragmentation is acquiring application information to determine a set of simple predicates. The simple predicates for relational data model take the form:

$$A_i \ \theta \ v_i$$

with $\theta \in \{=, \neq, <, \leq, >, \geq\}$ and A_i is a attribute of a relation schema $R = \{A_1, \ldots, A_n\}$ and $v_i \in dom(A_i)$. This format of simple predicates can only be used for the special case

when a class is defined only on a record type constructor and each attribute is defined only on a base type. For the object oriented data model simple predicates may be defined on identifiers as well as on values. According to Schewe & Thalheim [30] values can be grouped into types. The type system is a collection of types and can be defined by base types and constructors. The expression of simple predicates on values should be extended to suit various type constructors.

Simple predicates in the object oriented data model (OODM) can take a similar format to that in the RDM except that values can be complex, and the comparison operator θ can be used to compare sets. To define simple predicates we need also the path expressions which have been defined in Section 4.3.1.

Definition 6.1. Let C be a class. Simple predicates for class C have the form:

$$path_k \ \theta \ v_i$$

with $\theta \in \{=, \neq, \subset, \subseteq, \supset, \supseteq, \not\subset, \not\subseteq, \not\supset, \not\supseteq, \ni, \not\ni, \in, \notin\}$ and $path_k$ is a path expression for class C and v_i is a value of the type of the path, i.e. $v_i : T_{path_k}$.

The type of value v_i varies according to the type of path. Below are some situations that we might deal with:

(i) For a given class with structure expression $exp_C = \{exp\}$, a simple predicate has the form

$$value \ \theta \ v \quad \text{with} \quad v : T_C, \theta \in \{=, \neq, \subset, \subseteq, \supset, \supseteq, \not\subset, \not\subseteq, \not\supset, \not\supseteq\}$$

or

$$value \ \theta' \ v' \quad \text{where} \quad v' : T_{C'} \quad \text{with} \quad exp_{C'} = exp \quad \text{and} \quad \theta' \in \{\ni, \not\ni\}$$

(ii) If $r' : D$ appears in exp_C, i.e. $exp_C = \ldots, a_i : r : D, \ldots$, where D has structure expression exp_D, eg. $exp_D = (a_{i1} : exp_{i1}, \ldots, a_{in} : exp_{in})$, then the structure expression of class C can be represented with $exp_C = \ldots, a_i : exp_D, \ldots$. The representation type of class C is $T_C = \ldots a_i : T_D \ldots$. Then a simple predicate is in the form of

$$path \ \theta \ v_i \quad \text{where} \quad v_i : T_D \quad \text{and} \quad \theta \in \{=, \neq, <, \leq, >, \geq\}$$

(iii) If the representative type T_{path} of a *path* with expression exp_{path} is a base type b_i, we could use simple predicates $path\ \theta\ v_i$ and $v_i \in dom(b_i)$.

(iv) For a given class C with structure expression $exp_C = (a_1 : exp_1, \ldots, a_n : exp_n)$, the representative type is $T_C = (a_1 : T_1, \ldots, a_n : T_n)$. This case is analogue to a relational schema with a set of attributes $\{a_i, \ldots, a_n\}$.

(v) If in the class structure expression exp_C there is only a record type constructor without references involved and nesting is not deeper than 2, then simple predicates in the above definition can be simplified as:

$$value.a_i.a_{ij}\ \theta\ v_{ij}$$

where $v_{ij} \in dom(b_{ij})$, $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

Example 6.1. Choose class LECTURERC from the university database schema. `Class` LECTURERC IsA PERSONC `Struct` (PersonID: *NAT*, $\underbrace{\text{Name}}_{\varphi_1}$: ($\underbrace{\text{FName}}_{\varphi_2}$: *STRING*,LName: *STRING*, $\underbrace{\text{Title}}_{\varphi_3}$: {*STRING*}), $\underbrace{\text{Specialization}}_{\varphi_4}$: *STRING*, $\underbrace{\text{Department}}_{\varphi_5}$: DEPARTMENTC, Campus: *STRING*)

In example 4.5, we defined some path expressions on class LECTURERC. Using these path expressions some simple predicates on class LECTURERC can be defined as following:

(i) $\varphi \equiv ident = i_{101}$

(ii) $\varphi' \equiv value$ = (PersonID : 1001, (FName: John, LName:Dever, Title: {Professor, Dr})
Specialization: Pricing, Department: i_{403}, Campus: PN)

(iii) $\varphi_1 \equiv value$.Name = (FName: John, LName: Dever, Title: {Professor, Dr})
$\varphi_2 \equiv value$.Name.FName = 'John'
$\varphi_3 \equiv value$.Title $\not\subseteq$ {Professor, Dr}
$\varphi_4 \equiv value$.Specialization = 'Pricing'

(iv) $\varphi_5 \equiv value$!Department $= i_{403}$
$\varphi_5' \equiv value$!Department.Name = 'Marketing'

□

Example 6.2. Choose class SEMESTERC from the university database schema where structure expression of SEMESTERC is $exp_C = \{\underbrace{(\text{Year} : NAT, \text{Season} : NAT)}_{exp}\}$, there are some simple predicates defined on it:

$$value = \{(\text{Year} : 2000, \text{Season} : 02), (\text{Year} : 2000, \text{Season} : 12), (\text{Year} : 2002, \text{Season} : 1)\}$$

or

$$value \subseteq \{(\text{Year} : 2000, \text{Season} : 12), (\text{Year} : 2000, \text{Season} : 12)\}$$

or

$$value \ni (\text{Year} : 2000, \text{Season} : 12)$$

or

$$value \not\ni (\text{Year} : 2000, \text{Season} : 12)$$

□

6.2 Normal Predicates

The *minterm predicate* used in [24] is a conjunction that is in the form of

$$\mathcal{M} \equiv p_1^* \wedge \cdots \wedge p_n^*$$

where each p_i^* is either p_i or $\neg p_i$, $\{p_1, ..., p_n\} \subseteq \{A_i \theta v_i \mid A_i \in attr(R), \theta \in \{=, \neq, <, \leq, >, \geq\}$ and $A_i \theta v_i$ is a simple predicate defined on attribute A_i. Note this definition does not comprise all possible selection formulae. For these we have to request that φ is a disjunction (possibly infinite) of terms $(A_1 = v_1 \wedge \cdots \wedge A_m = v_m)$ with $R = \{A_1, \ldots, A_m\}$ and $v_i \in dom(A_i)$. The obvious generalization for the OODMSs is to choose a set of simple predicates for a class C instead of $\{p_1, \ldots, p_n\}$. Minterm predicates are only meaningful if they are satisfiable. Therefore, we introduce a new term of *normal predicates* on a class C as satisfiable minterm predicates. In this section we will define normal predicates for the object oriented data model (OODM) by adapting minterm predicates in the relational model.

Definition 6.2. Let $\Phi = \{\varphi_1, \ldots, \varphi_m\}$ denote a set of simple predicates on a class C. A set of normal predicates $\mathbb{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_n\}$ on class C is the set of all satisfiable predicates of the form:

$$\mathcal{N}_j \equiv \varphi_1^* \wedge \varphi_2^* \wedge \cdots \wedge \varphi_m^*$$

where φ_i^* is either φ_i or $\neg\varphi_i$.

Same as for the RDM the normal predicates do not exhaust all possible selection formulae on a class C. However, as the OODM allows cyclic references to be used on a schema, there are infinitely many such selection formulae. Even more, there are path expressions of arbitrary length, so we have no chance to capture the complete variety of selection. On the other hand, we want to define only finitely many fragments. So we do not lose much by restricting ourself to normal predicates.

Example 6.3. If there is a class CAMPUSSITEC with expression exp_C= *STRING*. A normal predicate can be defined as:

$$\mathcal{N} \equiv \underbrace{value = \text{'ALB'}}_{\varphi_1} \wedge \neg \underbrace{value = \text{'WN'}}_{\varphi_2}$$

□

Example 6.4. There is a class CARC with expression exp_C = (Brand : *STRING*, Age : *NAT*). A normal predicate which is a conjunction of simple predicates is defined as:

$$\mathcal{N} \equiv value.\text{Brand} = \text{'Ford'} \wedge value.\text{Age} \leq 9 \wedge value.\text{Age} > 6$$

where $\varphi_1 \equiv value.\text{Brand} = \text{'Ford'}$, $\varphi_2 \equiv value.\text{Age} \leq 9$ and $\varphi_3 \equiv value.\text{Age} > 6$ are simple predicates for a_i derived from expression exp_i. □

Example 6.5. Choose the schema of class PERSONC from our university schema, exp_C= (PersonID: *NAT*, Name:(FName: *STRING*, LName: *STRING*, Title: {*STRING*}) , Address: *STRING*, DOB: *DATE*)

A normal predicate on exp_{Title} = *STRING* is defined as:

$$\mathcal{N} \equiv \underbrace{value.\text{Name.Title} \subseteq \{\text{Professor, Dr}\}}_{\varphi_1} \wedge \underbrace{value.\text{Name.Title} \neq \varnothing}_{\varphi_2}$$

□

Example 6.6. Choose the following schema from our university schema:

`class` LECTURERC `structure` (PersonID: *NAT*, Name:(FName: *STRING*, LName: *STRING*, Titles: {*STRING*}), Specilization: *STRING*, Department: DEPARTMENTC, Campus: *STRING*)

Replace DEPARTMENTC with $exp_{\text{DEPARTMENTC}}$ = (Dname : *STRING*, Site : {(Campus : *STRING*, Dnumber : *NAT*)}, we get the following structure expression:

`class` LECTURERC `structure` (PersonID: *NAT*, Name:(FName: *STRING*, LName: *STRING*, Titles: {*STRING*}), Specilization: *STRING*, Department: (Dname: *STRING*, Site: {(Campus: *STRING*, Dnumber: *NAT*)}), Campus:*STRING*)

A normal predicate is defined as:

$$\mathcal{N} \equiv value!\text{Department.Dname} = \text{'IS'} \wedge value.\text{Campus} = \text{'PN'}$$

□

6.3 The Heuristic Fragmentation Process

The process of horizontal fragmentation for a class C in general includes the following steps:

(i) Determine a set of simple predicates Φ for C from queries that are most important and executed most frequently.

(ii) Partition class C into fragments according to the set of normal predicates $\mathcal{N}$ that are determined by the set Φ of simple predicates. The fragments obtained in this way will be disjoint and the original database can be reconstructed.

(iii) Merge fragments again by using union, if the merging reduces the costs.

(iv) In general, for each possible fragmentation that is defined by the union of these base fragments, there is a cost. We should choose the fragmentation with minimal costs. However, choosing a cost minimal fragmentation is a very complex optimization problem, which we cannot solve in general. In fact, we would have to consider all partitions of $\mathcal{N}$ for this, which is intractable. Therefore, we will explore a heuristic solution.

There are many factors that affect distribution design. Distribution design decisions are influenced by the logical organisation of the database, the location of the application requests, the access characteristics of the applications to the database, and the properties of the computer systems at each site. Information requirements for distribution design can be summarized in four categories [24]: database information, application information, communication network information and computer system information.

Database information such as global conceptual schema is needed at this stage. The size of each class instance can then be calculated. From the database schema, we can also know the relationship between classes, i.e. which class use other classes as references.

Application information, which is the description of user queries, updates and data access frequencies, is also needed at this stage. From application information, both quantitative and qualitative information that is needed in horizontal fragmentation can be acquired.

The fundamental qualitative information is:

- the ***simple predicates*** used in user queries [24]. According to the widely accepted "80/20 rule" [24, 6], in most practical situations, only the most active 20% user queries should be analysed because they usually account for 80% of the total data accesses.
- ***normal predicates*** can then be constructed with the set of simple predicates defined on classes. The formats of normal predicates have been defined in the above section.

Quantitative information about user applications, in the case of OODM includes:

- ***normal selectivity***, which is the number of objects of a class that will be selected by a user query according to a given normal predicate.
- ***access frequency***, which is the frequency with which user applications access data.

Communication network information, gives the communication cost factor. It also includes the information about the channel capacities, distances between sites and protocol overhead which is taken into consideration in some elaborate network models.

Computer system information is about the properties of the computer systems at each site. For each computer site, its storage and processing capacity should be known.

6.4 A Cost Model

Whether a design of horizontal fragmentation will improve the system performance depends on whether the total execution query cost will be decreased significantly. Therefore, a cost model is needed to evaluate a fragmentation schema. To calculate query costs we need a query tree. We also need some formal formulae to calculate the sizes of classes, fragments, and intermediate nodes of a query tree.

6.4.1 Query-Trees

Suppose we are given a query in query algebra. While query optimization is beyond the scope of this thesis, we may assume the query is optimized. In order to calculate query costs, we adapt query trees from [28]. A query tree is just a graph representation of a query.

The leaves of a query tree will be elementary queries such as classes C or fragments f_i. All other nodes are operators of query algebra: σ_φ, π_X, ρ_f, $\bowtie$, $\cup, \cap$, or $-$ introduced in Section 4.3.2 . The query tree is formed inductively (inside-out) from Q as follows:

- If Q is elementary, then the tree consists of just one root, which is Q.
- For $Q = op(Q')$ with op standing for a selection, projection or renaming take the whole tree for Q' plus a new root op having the root of the Q'-tree as its successor.
- For $Q = Q_1 op Q_2$ with op standing for a join, a union or a difference take the trees for Q_1 and Q_2 plus a new root op having the roots of the Q_1-tree and the Q_2-tree as successors.
- If the fragmentation of class C leads to

$$db(C) = \bigcup_{i=1}^{n} db(C_i) \quad \text{with disjoint sets} \quad db(C_i) = \sigma_{\varphi_i}(db(C)), \quad 1 \leq i \leq n,$$

  then replace C in each query Q by $C_1 \cup \ldots C_n$

  If the fragmentation of class C leads to

$$db(C) = (\ldots((db(C_1) \bowtie_{t_{1,2}} db(C_2)) \bowtie_{t_{1,2,3}} \ldots) \bowtie_{t_{1,\ldots,k}} db(C_k))$$

where $db(C_i) = \pi_{X_i}(db(C))$ with $X_i = \{a_1^i, ..., a_{n_i}^i\}$, $t_{1,...,k} = (a_{x_1^i} : t_{x_1^i}, ..., a_{x_{l_i}^i} : t_{x_{l_i}^i})$ with $(a_{x_1^i}, \ldots, a_{x_{l_i}^i}) = (X_1 \cup ... \cup X_{i-1}) \cap X_i$ and t_x is the representation type for the structure expression exp_x. Then replace class C by

$$(\ldots((C_1 \bowtie_{t_{1,2}} C_2) \bowtie_{t_{1,2,3}} \ldots) \bowtie_{t_{1,\ldots,k}} C_k)$$

Example 6.7. Suppose class LECTURERC has been horizontally fragmented into three classes as we did in Example 5.2 . Then we get $db(\text{LECTURERC}) = db(\text{LECTURERC}_1) \cup db(\text{LECTURERC}_2) \cup db(\text{LECTURERC}_3)$. If we want know the students that have a lecturer in the Department of Information Systems as a supervisor, then we have the following query:

$$\pi_{\text{StudentID, LName, FName}}(\sigma_{value.\text{Department='IS'}}(\text{LECTURERC}) \bowtie \text{STUDENTC}))$$

The corresponding query is shown in Figure 6.1

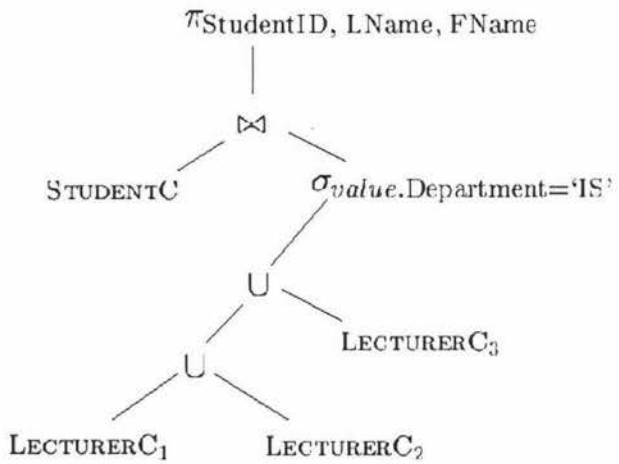


Figure 6.1: Example of a Query Tree

6.4.2 Calculation of Size for Classes

The calculation of the size of classes is more complicated in the OO model than the calculation of size for relations in the relational model. In the relational model, we use only the record type constructor. Then the size of a relation can be calculated based on the size of a tuple. In the object oriented model, however, there are other type constructors including set and maybe some other bulk type constructors. Therefore, at least the size calculation for

classes should be discussed in different type constructors. The size calculation for classes on record type constructors will be treated as a special case in the object oriented environment. The paragraph below will extend the theory in Schewe [28] and discuss the calculation of class size in the object oriented database environment.

(i) Base Types

Let us first consider the simplest situation: a class C with a representation type $T_C = b$, with b indicating a base type.

Assumptions:

- Let n denote the number of objects in a class instance $db(C)$ over T_C.
- Let l_v denote the average space (in bits) for a value of type b.
- Let l_{id} denote the size of an identifier, i.e. a value of type ID.

Then the average size of $db(C)$ is

$$s_C = n \cdot (l_v + l_{id})$$

Note that this basic case already includes references, which lead to the type $b = ID$. Thus $l'_v = l_{id}$.

(ii) Record Type Constructor

Now let C be a class defined by a record type constructor, i.e. we have $T_C = (a_1 : T_1, \ldots, a_n : T_n)$. The calculation of the size of a class is very similar to that in the relational model.

Assumptions:

- Let n denote the number of objects in a class instance $db(C)$ over T_C.
- $l_0 = l_{id}$ and l_{id} denote the size of an identifier, i.e. a value of type ID.
- Let l_j denote the average space (in bits) for attribute a_j of a class C.

The average size of $db(C)$ over exp_C is:

$$s_C = n \cdot \sum_{j=0}^{k} l_j$$

(iii) Set Constructor

If there is a class C with a structure expression $T_C = \{exp\}$, the calculation of the size of each class instance is different from the above.

Assumptions:

- Let n denote the number of objects in a class instance $db(C)$ over T_C.
- Let l_{id} denote the size of an identifier, i.e. a value of type ID.
- Let m denote the number of values of an object o in a class instance $db(C)$ over T_C.
- Let l_v denote the average space (in bits) for a value over exp of an object o in a class instance.

The average size of a object o over T_C is

$$l_{id} + m \cdot l_v$$

and the average size of class over T_C is

$$s_C = n \cdot (l_{id} + m \cdot l_v)$$

Example 6.8. We calculate the size of db(LECTURERC) over the university database schema. The structure expression of class LECTURERC is `class` LECTURERC `structure` (PersonID: *NAT*, Name:(FName: *STRING*, LName: *STRING*, Titles: {*STRING*}), Specilization: *STRING*, Department: DEPARTMENTC, Campus: *STRING*).

We assume:

(i) the average number n of objects in a class instance db(DEPARTMENTC) is 200,

(ii) the largest number for id in the instance of the database is 100000, then $l_0 = l_{id} = 17$ bits.

We first determine the average space (in bits) l_j for attribute a_j of a class LECTURERC:

- domain of PersonID is $NAT(4)$, the biggest number is 9999, hence we get $l_{\text{PersonID}} =$ 13bits,
- calculate the size of Name:
  - domain of FName is $STRING(20)$, mean length is estimated 12, hence $l_{\text{FName}} = 12 \cdot 8 = 96$,
  - domain of LName is $STRING(20)$, mean length is 12, then $l_{\text{LName}} = 12 \cdot 8 = 96$,
  - size of Titles is the size of a set of values, assume the average number of titles of a person is 2, domain of Title is $STRING(15)$ with average length as 10, $l_{\text{Title}} = 2 \cdot 10 \cdot 8 = 160$,

  Then the size of Name is $l_{\text{Name}} = 96 + 96 + 160 = 352$,
- domain of Specialization is $STRING(40)$ with average size 20, then $l_{\text{Specialization}} = 20 \cdot 8 = 160$,
- value of Department is an id of a department, hence $l_{id} = 17$,
- domain of Campus is $STRING(20)$ and the mean length is 10, hence $l_{\text{Campus}} = 10 \cdot 8 = 80$.

The average size of a object in $db(\text{LECTURERC})$ over $exp_{\text{LecturerC}}$ is:

$$l_{\text{LECTURER}} = \sum_{j=0}^{k} l_j = 17 + 13 + 352 + 160 + 17 + 80 = 639$$

Therefore, the average size of an instance of class LECTURERC over $exp_{\text{LecturerC}}$ is:

$$s_{\text{LECTURERC}} = n \cdot \sum_{j=0}^{k} l_j = 200 \cdot 639 = 127800$$

□

The cost model is still not realistic. Even if we accept the inaccuracy resulting from using only mean values, it is hardly the case that a complex value will be stored as a simple unit,

especially if sets are involved. It is more likely that data structures known from the network data model will be used for this.

If T_C involves a set type constructor, i.e. $T_C = (a_1 : T_1, \ldots, a_i : \{T_i\}, \ldots, a_n : T_n)$, we would store values of type T_i separately as a linked list and just include a pointer to the first element of this list as well as a pointer back from the last element of the list. See Figure 6.5 for an illustration.

$$T_C = (a_1 : T_1, \ldots, a_i : \{T_i\}, \ldots, a_n : T_n)$$

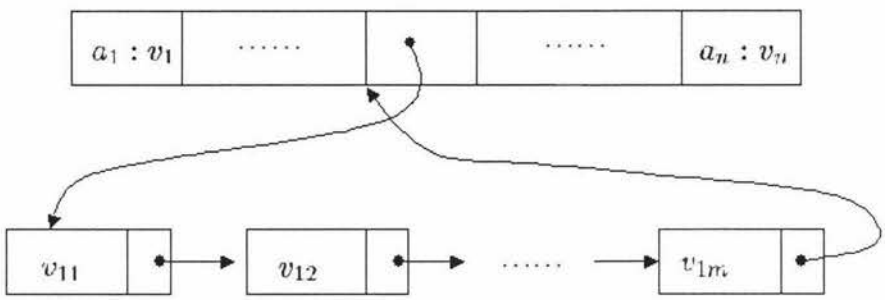


Figure 6.2: Data Structure of Type T_C

This means that instead of l_v we have to consider $l_{id} + l_v$ assuming the l_{id} is the size of a pointer. Then the whole set value needs the space

$$l_{id} + m \cdot (l_{id} + l_v)$$

where m is again the average number of elements in the set. Note that the size of a set is $m \cdot l_v + (m+1) \cdot l_{id}$. It means that we need $(m+1) \cdot l_{id}$ more space to store the values of a set if we treat it as being stored in a linked list with a pointer to an object o of a class C where m is the number of objects in the set.

Example 6.9. Making the same assumptions, we recalculate the size of db(LECTURERC) in the above example by considering values of Titles stored as a linked list with a pointer in a value of type $T_{\text{LECTURERC}}$. Hence, the size of Titles is:

$$l'_{\text{Titles}} = l_{id} + m \cdot (l_{id} + l_v) = 17 + 2(17 + 80) = 211$$

The size of an object in db(LECTURERC) is:

$$l'_{\text{LECTURER}} = l_{\text{LECTURER}} + (m+1) \cdot l_{id} = 639 + (2+1) \cdot 17 = 690$$

Therefore, the size of instance $db(\text{LECTURERC})$ of class LECTURERC is:

$$s_{\text{LECTURERC}} = n \cdot l'_{\text{LECTURER}} = 200 \cdot 690 = 138000$$

□

6.4.3 Calculation of Size for Fragments or Intermediate Nodes

Once a class has been horizontally fragmented, each fragment contains a subset of the objects of the original class. The size of a horizontal fragment can be calculated according to the selectivity of the selection operation that is used to obtain the fragment.

If there is a set of selection formulae $\{\varphi_1, \ldots, \varphi_r\}$ defined on class C, and $100 \cdot p_i$ is the average percentage of objects in $db(C)$ satisfying $\varphi_i (i = 1, \ldots, r)$ with $\sum_{i=1}^{r} p_i = 1$, then the average size of horizontal fragments over $db(C_i) = \sigma_{\varphi_i}(db(C))(i = 1, \ldots, r)$ is:

$$p_i \cdot s_C$$

The calculation of s_C is discussed in the previous subsection.

Example 6.10. Continuing example 6.9, if there is a set of selection formulae $\{\varphi_1, \varphi_1, \varphi_3\}$ defined on class LECTURERC:

$$\varphi_1 \equiv \textit{value}!\text{Department.Name} = \text{'Marketing'}$$

$$\varphi_2 \equiv \textit{value}!\text{Department.Name} = \text{'Information Systems'}$$

$$\varphi_3 \equiv \textit{value}!\text{Department.Name} = \text{'Accounting'}$$

Assume there are only three department names in the database, $p_1 = 0.3$, $p_2 = 0.4$ and $p_3 = 0.3$, and $\sum_{i=1}^{3} p_i = 1$.

The average sizes of horizontal fragments from $db(\text{LECTURERC}_i) = \sigma_{\varphi_i}(db(\text{LECTURERC}))$ are:

$$s_{\text{LECTURERC}_1} = p_1 \cdot s_{\text{LECTURERC}} = 0.3 \cdot 138000 = 41400$$

$$s_{\text{LECTURERC}_2} = p_2 \cdot s_{\text{LECTURERC}} = 0.4 \cdot 138000 = 55200$$

$$s_{\text{LECTURERC}_3} = p_3 \cdot s_{\text{LECTURERC}} = 0.3 \cdot 138000 = 41400$$

□

For a query tree we need to calculate the size of intermediate nodes besides leaves. In a query tree, intermediate nodes are some operations defined on fragments or classes. We use s to denote the size of the successor of an operation. We adapt the calculation formulae from Schewe [28] to the object oriented databases to get the following formulae:

- The size of a projection node π_{exp} is $(1 - c_i) \cdot s \cdot \frac{l_f}{l_o}$ where $l_f(l_o)$ is the average size of an object over $exp(exp_C)$. s is the size assigned to the successor and c_i is the probability that two classes coincide on exp.

- For a join node the assigned size is $\frac{s_1}{l_1} \cdot p \cdot \frac{s_2}{l_2}(l_1 + l_2 - l)$, where s_i are the sizes of the successors, l_i are the corresponding object sizes, l is the size of a tuple over the common attributes and p is the matching probability.

- For a union node the size is $s_1 + s_2 - p \cdot s_1$ with the probability p for an object of C_1 to coincide with an object of C_2.

- For a difference node the assigned size is $s_1 \cdot (1 - p)$ with the probability p for an object of C_1 to coincide with an object of C_2.

- For a renaming node the assigned size is exactly the size s assigned to the successor.

6.4.4 Allocate Intermediate Nodes to Sites

Fragmentation of a class C results in a set of fragments $\{f_1, \ldots, f_n\}$ of average sizes $s_1, \ldots, s_n$. The network has nodes $N_1, \ldots, N_k$. The allocation problem is to allocate each fragment f_i to one of the sites N_j. Therefore, allocation assignment λ for a query tree assigns a node $N_j (1 \leq j \leq k)$ to each node in the query tree. Fragment allocation is a mapping:

$$\lambda : \{1, \ldots, n\} \rightarrow \{1, \ldots, k\}$$

Each leaf fragment f_1 is assigned the node to which the fragment is allocated. The root Q of the tree is assigned the node at which the query is issued.

Example 6.11. Continue Example 6.7 by allocating one of the sites to each of the fragments as well as to other leaves and the root. Suppose the network has 3 nodes and the query is executed at site 1, then the root of the tree is assigned site 1. Figure 6.3 gives a example of a query tree with locations.

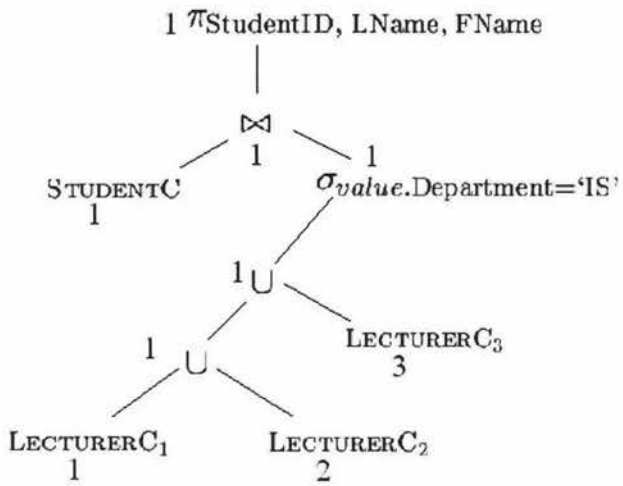


Figure 6.3: Query Tree with Locations

□

Site allocation should be done after we calculate the sizes of all leaves, intermediate nodes and the root of a query tree. If a node E has successor nodes that are evaluated at site N_1 and N_2, respectively, then E should be assigned to either site N_1 or N_2, preferably the site where the least data must be transferred. It is impossible to allocate other sites besides the site of its successors.

6.4.5 Calculation of Query Costs

After a class has been fragmented, the fragments of the class will be allocated to a location. Ideally, we should choose a site that is closest to the site that issues the bulk load of queries which access the fragment. However, this may not lead to the optimum query processing time.

Once we know the size of horizontal fragments by applying the formulae described in the

previous section, we can calculate the query costs for any allocation. Again, minimizing the total query cost is computationally intractable, so we have to look for a heuristic solution.

The cost calculation formula proposed in [26] can be easily adopted to the object oriented case. For each class C_i, there is a set of queries $Q^m = \{Q_1, \ldots, Q_m\}$ accessing it. Under a certain location assignment λ, for a given query Q_j, query costs are composed of two parts: *storage costs* and *transportation costs*. The formula below shows the components of the query costs:

$$costs_\lambda(Q_j) = stor_\lambda(Q_j) + trans_\lambda(Q_j)$$

The storage costs of a query Q_j depend on the size of the involved classes or fragments, and on the assigned locations which decide the storage cost factors. It can be expressed by the following formula:

$$stor_\lambda(Q_j) = \sum_h s(h) \cdot d_{\lambda(h)}$$

where h ranges over the locations (sites) of the distributed database system, $s(h)$ are the sizes of involved classes or the fragments of some classes and $d_{\lambda(h)}$ indicates the storage cost factors.

The transportation costs of query Q_j depend on the sizes of the involved classes or fragments of classes and on the assigned locations which decide the transport cost factor between every pair of sites. It can be expressed by the following formula:

$$trans_\lambda(Q_j) = \sum_h c_{\lambda(h)\lambda(h')} \cdot s(h)$$

where $c_{\lambda(h)\lambda(h')}$ are the transportation cost factors.

For each query Q_j we get a value for its frequency $freq_j$. The total costs of all the queries in Q^m are the sum of the costs of each query multiplied by its frequency. It can be expressed by the following formula:

$$cost_{\lambda m}(total) = \sum_{j=1}^{m} cost_{\lambda m}(Q_j) \cdot freq_j$$

6.5 A Heuristic Procedure for Horizontal Fragmentation

Fragmentation and allocation are considered as two isolated problems in [24]. After fragmentation, the fragments are allocated to reside at one node in a distributed management system (not considering replication at this stage in this thesis). The design of fragmentation schemata on relations in [24] did not use a cost model. But we argue that the values of the costs of queries after fragmentation will affect the decision on whether we need to perform fragmentation or not. A cost model should be used to evaluate different fragment solutions. Let us look at the following example to see whether fragmentation of a class and allocating result fragments to different sites will achieve better performance.

Example 6.12. Consider a class being fragmented into two fragments f_1, f_2, and two queries Q_1, Q_2 executing at two different sites N_1, N_2 to access these two fragments remotely. The frequencies of Q_1 and Q_2 are $freq_1$ and $freq_2$, respectively. This design is shown in Figure 6.4.

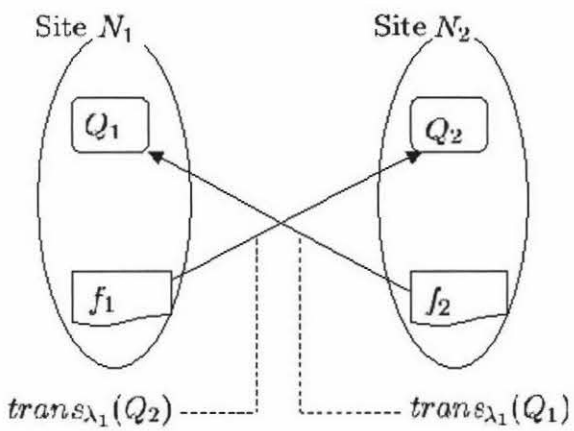


Figure 6.4: Allocation of Fragments to Two Sites

If the sizes of f_1 and f_2 are s_1 and s_2, respectively, ignoring storing costs for all the fragments, we have total query costs of:

$$cost_{\lambda_1} = trans_{\lambda_1}(Q_1) + trans_{\lambda_1}(Q_2) = s_2 \cdot freq_1 \cdot c_{21} + s_1 \cdot freq_2 \cdot c_{12}$$

with c_{12} and c_{21} as transportation cost factors. Generally, c_{12} should be equal to c_{21}.

If query Q_1 is executed more frequently, say $freq_1 > freq_2$, we do not fragment the class and put the whole class at site N_1, the site that Q_1 is executed. This design is shown in Figure 6.5.

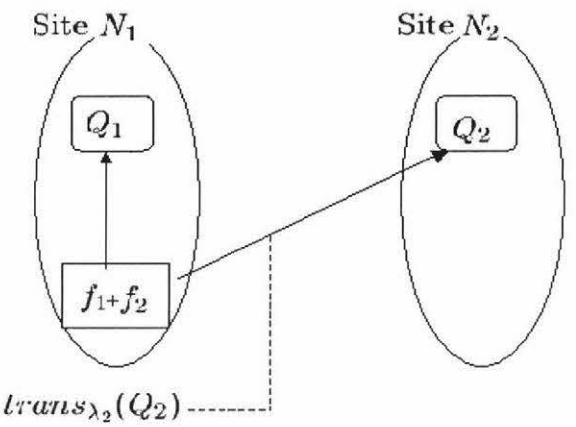


Figure 6.5: Allocation of Class to One Site

The total query costs for the second design are:

$$cost_{\lambda_2} = trans_{\lambda_2}(Q_2) = (s_1 + s_2) \cdot freq_2 \cdot c_{12}$$

Comparing costs $cost_{\lambda_1}$ and $cost_{\lambda_2}$ we get $cost_{\lambda_1} > cost_{\lambda_2}$. □

It can be concluded that fragmentation of classes does not always minimize the query cost. It is an NP-hard problem to find the optimized fragmentation solution by computing total costs for all possible fragmentation schema [24]. In this section we propose a heuristic approach for identifying a reasonable degree of fragmentation that leads to relatively minimal total query costs.

For a given database schema $\mathcal{S} = \{C_1, \ldots, C_i, \ldots C_n\}$, there is a set of queries $Q^m = \{Q_1, \ldots, Q_j, \ldots, Q_m\}$ that access the database most frequently or that are used by the most critical transactions.

The heuristic procedure of horizontal fragmentation should include the following steps:

1. Identify the set of most frequent queries and rewrite these queries in the form of the algorithm introduced in Section 4.3.

2. Identify the sites at which the queries will be issued. Treat queries that are started at several sites as (several) different queries. Estimate for each query the frequency of its execution.

3. Sort queries by their frequencies. This provides a list of queries $Q^m = [Q_1, \ldots, Q_j, \ldots, Q_m]$ with the corresponding list of values of frequencies $F^m = [freq_1, \ldots, freq_m]$ for Q^m where we always get $freq_{ik-1} \geq freq_{ik}$.

4. Determine optimized query trees for all of the queries. Extract simple predicates from these trees.

5. Construct a usage matrix based on the simple predicates obtained in the previous steps. We get a set Φ of simple predicates φ.

	Q_1	$\ldots$	Q_m
C_1	φ_{11}	$\ldots$	$\ldots$
C_2	φ_{21}	$\ldots$	$\ldots$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
C_n	φ_{n1}	$\ldots$	φ_{nm}

6. From the matrix obtained in the previous step we can get a list Φ_i^b of sorted simple predicates $\Phi_i^b = [\varphi_{i1}, \ldots \varphi_{ib}]$ where we always get $freq_{i-1} \geq freq_i$. The number of simple predicates is b. We get a list $X = [0, 1, \ldots, x_1, \ldots, x_2, \ldots, b]$ of indices for the simple predicates.

7. Perform the following steps iteratively to find a reasonable number of simple predicates for horizontal fragmentation and fragment the given class C_i.

   Choose four numbers a, b, x_1, x_2 from X with $a < x_1 < x_2 < b$, including initially the two bounds, i.e. start with $0, x_1, x_2, b$ with $0 < x_1 < x_2 < b$. For each number $x \in \{a, b, x_1, x_2\}$ we calculate the corresponding query costs by the following procedures:

   (I) Choose first x simple predicates in the list Φ_i^b, i.e. $\Phi_i^x = \{\varphi_{i1}, \ldots \varphi_{ix}\}$ and build the corresponding set of normal predicates $\mathcal{N}^x$.

   (II) Fragment the class C_i according to the set $\mathcal{N}^x$ of normal predicates obtained in the previous step. Let the fragments be $F_1^x, \ldots, F_r^x$.

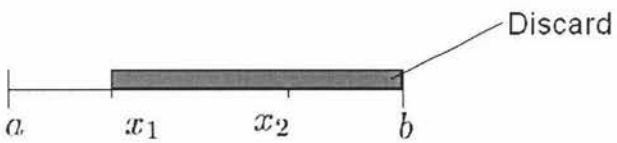
(III) Calculate the frequencies $freq_k^j$ of access to the fragments F_k from site j, and use this to determine fragment allocation to sites. Put the fragments to the nodes that access them most frequently.

(IV) Calculate the total query costs $cost_x$ using the cost model introduced in Section 6.4.5.

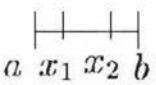
We get four values for the query costs $cost_a, cost_{x_1}, cost_{x_2}, cost_b$. Let $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b)$ denote the minimal value among the four values. Comparing these query costs we might have the following four situations:

- If $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b) = cost_a$ then set $b = x_1$ and choose two new values for x_1, x_2 satisfying $a < x_1 < x_2 < b$. If we can find two such numbers then continue the procedure.

  if $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b) = cost_a$

  then

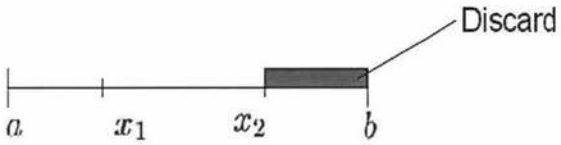
  If we can only obtain one number x from the list X, calculate the query cost for it. If $cost_x < cost_a$, then x should be the number of simple predicates that we should use to fragment the class C_i. Thus, let $y := x$.

  If $cost_x > cost_a$ or there is no number left between the new a and b, then a is the number of simple predicates that we will use for fragmentation. Thus, let $y := a$.

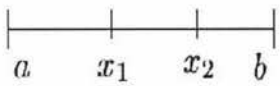
- If $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b) = cost_{x_1}$, then set $b = x_2$ and choose a new number between a and b so that we get new numbers for x_1, x_2. Continue with the procedure.

  If we cannot find a new number between new a and b, then x_1 is the number of simple predicates that we should choose for fragmentation. Thus $y := x_1$.

if $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b) = cost_{x_1}$



then



- If $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b) = cost_{x_2}$, then set $a = x_1$. Choose a new number between a and b so that we get new values for x_1, x_2 and continue the procedure with the new $a < x_1 < x_2 < b$. If we do not find such a new number, x_2 will be the number of simple predicates that we need for fragmenting the class C_i, i.e. take $y := x_2$.
- If $min(cost_a, cost_{x_1}, cost_{x_2}, cost_b) = cost_b$, then set $a = x_2$ and find two new numbers between a and b satisfying the condition $a < x_1 < x_2 < b$. If we can find such numbers, then we continue the iteration with the new values.

  If there is only one number x left between a and b, we calculate the corresponding query cost and compare it with $cost_b$. The number that leads to the minimal cost will be the number of simple predicates that we need for horizontal fragmentation, i.e. for $cost_x < cost_b$ we choose $y := x$, otherwise $y := b$.

  If there is no number between a and b left in X, then b is the number of simple predicates needed for fragmenting C_i. Thus, $y := b$.

The result is the set $\Phi_i^y = \{\varphi_{i1}, \ldots \varphi_{iy}\}$ of simple predicates. Take the corresponding set $\mathcal{N}^y$ of normal predicates for fragmenting C_i.

Note that a, b, x_1, x_2 are updated in each loop. Searching requires only one or two new points in each loop. Query costs have to be calculated only for these new points. The procedure stops when the remaining interval (a, b) contains only one number or is empty.

The above procedure is a heuristic one based on the assumption that a reasonable fragmentation schema can be obtained by looking at most frequently used simple predicates. The

approach presented above can rapidly search for a reasonable number of simple predicates and results in presumably low total query costs.

6.6 Summary

In this chapter horizontal fragmentation for object oriented databases has been discussed in more detail. Definitions for simple predicates and normal predicates have been outlined. A heuristic horizontal fragmentation procedure has been proposed based on a cost model. The horizontal fragmentation approach presented in this chapter is superior to that presented in Özsu & Valduriez [24] and the one by Ezeife & Barker [13]. Characteristics and potential benefits of the approach presented in the chapter, and differences to other approaches in the literature can be summarized as follows:

Firstly, the approach can deal with both simple and complex attributes using the same procedure. In [13], simple and complex attributes are treated with different algorithms. Simple attributes refer to the attributes of primitive attribute types only, i.e. those that do not contain other classes as part of them. Complex attributes have the domain of attribute as another class [13]. In our approach, attributes are defined using the underlying object oriented type system. Attributes defined on the abstract identifier type ID are treated as being defined on one of the base types. Our approach is more universal than others in terms of dealing with attributes of different types, and is more easily put into practical use.

Secondly, the expression of simple predicates is extended such that simple predicates can be defined on identifiers as well as on values. The expression of simple predicates on values is also extended to suit various type constructors in the underlying type system of the object oriented model. In Ezeife & Barker [13], the definition of simple predicates is adopted from [24] without any adaption. However, the format of simple predicates in [24] cannot deal with the situation that a simple predicate is defined on a complex type constructor, i.e. a finite set type constructor. In the approach presented, the format of simple predicates is extended in a way that simple predicates can be defined not only on record type constructors or base type constructors but also on finite set type constructors. Note that the domain of the values for the comparison operator θ has been extended to include some set comparing

operations. Path expressions have been introduced to refer to any elements of a class.

Thirdly, instead of using minterm predicates as found in [24] and [13], we introduced normal predicates on classes as the satisfiable minterm predicates. Horizontal fragmentation operations based on a set of normal predicates could therefore be guaranteed to satisfy the characteristics of fragmentation discussed in Section 3.2. The approach did not rely on dependencies between simple queries as in [24] , because these can hardly be determined. It is very hard to use these dependencies to determine the satisfiability of a conjunction of simple predicates, or to simplify them if they are satisfiable.

Fourthly, the approach applies a cost model for horizontal fragmentation design. In [24], horizontal fragmentation is performed without evaluating the overall system performance. We argue that the larger degree of fragmentation does not necessary lead to the better system overall performance. There exists a cut off point for the degree of horizontal fragmentation that the system has the best performance. However it is computationally intractable to find the optimized fragmentation solution by comparing total costs for all possible fragmentation schemata [24]. The heuristic procedure proposed in this chapter is based on a cost model with which the system performance can be evaluated once a database is being fragmented. One of the characteristics of this procedure is that it can rapidly achieve a horizontal fragmentation schema that is designed to result in low total query cost, or, in other words, the system's overall performance being improved.

Our approach is based on a rather sophisticated data model and the assumption that a reasonable fragmentation schema can be obtained by looking at most frequently used simple predicates. This is the first attempt to investigate horizontal fragmentation for object oriented databases in detail. The obtained horizontal fragmentation procedure can be used to assist designers for the distribution design for object oriented databases.

Chapter 7

Conclusion and Possible Extension

7.1 Summary

Fragmentation is one of the distribution design techniques that must be used to set up distributed database systems. A lot of research has been done on fragmentation techniques for distributed databases. However, most of it refers to the relational data model and very few researchers refer to the object oriented data model (OODM). Fragmentation of object oriented databases (OODBs) is much more complex than fragmentation in the relational data model due to the fact that the semantic model of object oriented databases is much richer than that of relational databases. This thesis studies the critical adaptation and generalisation of existing fragmentation techniques for relational databases for OODBs.

The major focus of the thesis is on the horizontal fragmentation of classes in OODBs. For this I first reviewed important basic concepts and characteristics of distributed databases. I further reviewed an OODM and introduced path expressions and queries for it. Based on existing horizontal fragmentation techniques for relational databases, I introduced normal predicates for the object oriented model, which are satisfiable conjunctions of simple predicates. The normal predicates are used for the horizontal fragmentation of OODB schemata.

The degree of horizontal fragmentation was shown to affect the performance of distributed database systems. A high degree of horizontal fragmentation does not always lead to performance improvements of distributed database systems. Previous horizontal fragmentation

approaches did not deal in any depth with evaluating the results of horizontal fragmentation with respect to anticipated performance improvement. This thesis argues that a cost model is required for making decisions on the degree of fragmentation. Based on the assumption that a reasonable fragmentation schema can be obtained by looking at most frequently used simple predicates, a heuristic procedure for horizontal fragmentation of OODBs is proposed. This heuristic approach is based on a query cost model. While, a global cost optimism can not be guaranteed. An important characteristic of the approach presented is that it can rapidly search for a reasonable number of simple predicates which results in presumably low total query costs.

7.2 Future Work

The complexity of the overall research problem addressed in this thesis resulted in a number of assumptions and constraints. Based on the work and results presented, extensive more research can and needs to be undertaken to make the whole database design theory for object oriented databases solid and complete. Research results obtained in this project can be improved and extended in various ways including:

- Practical evaluation of the proposed procedure

  It is essential to verify the claims that the proposed procedure for horizontal fragmentation reduces total query costs. Therefore, we need to implement this procedure, run a number of experiments and evaluate their results. Based on this practical evaluation, optimization or refinement of the procedure can be proposed.

- Extending the underlying type system

  The object oriented data model applied in this thesis is a simplified data model of that introduced in [31]. The object oriented data model introduced in [31] is based on a type system that includes list and bag type constructors. With the list and bag type constructors, the object oriented data model can deal with more complex data types. Inclusion of these two type constructors will make the proposed fragmentation operations more universal.

- Object methods

  For the sake of simplicity we did not take into account methods when we discussed fragmentation of classes. As mentioned in Chapter 4, a class has a set of methods. It is more general to discuss fragmentation for object oriented databases by taking into account methods. Horizontal fragmentation of a class results in a set of new classes such that each of the new classes has the same structure as its original class. Therefore all methods defined for the original class should also be defined for each of the new classes. Attention need to be paid to the situation that a class has methods that access other classes.

- Transportation cost factors

  In our cost model, transportation cost factors are identified as statistical numbers which are used to measure communication costs between each pair of nodes on the network. In a real-life situation, these cost factors might change over time. Thus, additional analysis is needed to determine current values of all related transportation cost factors.

- Vertical and splitting fragmentation

  In this project, we generalize three fragmentation operations for object oriented databases: horizontal, splitting and vertical fragmentation. While the first, horizontal fragmentation, has been investigated in detail, the work needs to be extended regarding vertical and splitting fragmentation. For vertical fragmentation, bond energy algorithm (BEA) is used in relational databases to cluster attributes according their pairwise affinity [16]. While we aim to carry forward those results, it is most likely that this extension will be done to a simplified OODM first, with further extension added later.

- Allocation and replication of objects

  So far, we neglected replication of objects. However, replication is an important technique to improve the reliability and availability of a distributed database system. Therefore the query cost model should be adjusted to reflect this fact. It is noted that the data transmission overhead for update and that for retrieval requests are different.

In an update query, it is necessary to inform all nodes that replicas exist, while in a retrieval query, only one node that holds one of the copies should be informed [24]. Further, the data should be locked while it is accessed by a update query, while the data accessed by a retrieval query does not need to be locked. Allocation of fragments involves an allocation model that attempts to minimize the total cost of processing and storage while trying to meet certain response time restriction [24]. It is announced that the allocation problem is an NP-complete problem. The generic allocation model [24], which is used for the relational model, is too complex to be developed. Therefore, for the object oriented model heuristic methods that yields suboptimal solutions will be sought.

- Building prototypes for further research results

  It is expected that an experimental evaluation should be applied when further research results have been obtained.

Bibliography

[1] P. M. G. Apers, *Data allocation in distributed database systems*, ACM Trans. Database Syst. **13** (1988), 263–304.

[2] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, and S. Zdonik, *The object-oriented database system manifesto*, Proc. First International Conference on Deductive and Object Oriented Database (Kyoto, Japan) (S. Nishio W. Kim, J.-M. Nicolas, ed.), December 1989, pp. 40–57.

[3] S. Atre and S. Advisor, *Distributed databases, cooperative processing & networking.*, McGraw-Hill, 1992.

[4] D. Bell and J. Grimson, *Distributed database system*, Addison-Wesley, 1994.

[5] L. Bellatreche, K. Karlapalem, and A. Simonet, *Horizontal class partitioning in object-oriented databases*, Database and Expert Systems Applications, 1997, pp. 58–67.

[6] S. Ceri and G. Pelagatti, *Distributed databases principles and system*, McGraw-Hill, New York, 1984.

[7] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. Navathe, *An objective function for vertically partitioning relations in distributed databases and its analysis*, Tech. Report UF-CIS-TR-92-045, University of Florida, Gainesville, FL, 1992.

[8] P.-C. Chu, *A transaction-oriented approach to attribute partitioning*, Information Systems **17** (1992), no. 4, 329–342.

[9] E. F. Codd, *A relational model for large shared data banks*, Commun. ACM **13** (1970), no. 6, 377–387.

[10] ———, *Relational databases: A practical foundation for productivity*, Commun. ACM **25** (1982), no. 2, 109–117.

[11] D. W. Cornell and P. S. Yu, *A vertical partitioning algorithm for relational databases*, Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA, IEEE Computer Society, 1987, pp. 30–35.

[12] H. Darwen and C. J. Date, *The third manifesto*, ACM SIGMOD Record **24** (1995), no. 1, 39–49.

[13] C. I. Ezeife and K. Barker, *A comprehensive approach to horizontal class fragmentation in a distributed object based system*, Tech. report, Advanced Database Systems Laboratory, Department of Computer Science, University of Manitoba, Canada, October 1994.

[14] B. Gavish and H. Pirkul, *Computer and database location in distributed computer systems*, IEEE Transactions on Computers **C-35** (1986), no. 7, 583–590.

[15] V. Gligor and R. Popescu-Zeletin, *Transaction management in distributed heterogeneous database management system*, Journal of Information Systems **11** (1986), no. 4, 287–297.

[16] J. A. Hoffer and D. G. Severance, *The use of cluster analysis in physical database design*, Proc. First International Conference on Very Large Data Bases (Framingham, MA), Septemper 1975.

[17] K. Karlapalem and S. B. Navathe, *Materialization of redesigned distributed relational databases*, Master's thesis, Hong Kong University of Science and Technology, Hong Kong, 1994.

[18] G. McFarland, A. Rudmik, and D. Lange, *Object-oriented database management systems revisited*, Tech. Report SP0700-98-4000, The Data & Center for Software, Indialantic, 1999.

[19] J. Muthuraj, *A formal approach to the vertical partitioning problem in distributed database design*, Master's thesis, The University of Florida, Florida, USA, 1992.

[20] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dour, *Vertical partitioning algorithms for database design*, ACM TODS **9** (1984), no. 4, 680–710.

[21] S. B. Navathe, K. Karlapalem, and N. Ra, *A mixed fragmentation methodology for initial distributed database design*, Tech. Report TR 90-17, CIS Dept, Univ. of Florida, Gainesville, FL, 1990.

[22] S. B. Navathe and M. Ra, *Vertical partitioning for database design: A graphical algorithm*, ACM SIGMOD **14** (1989), no. 4, 440–450.

[23] M. T. Özsu and P. Valduriez, *Distributed database systems: Where are we now?*, IEEE Computer **24** (1991), no. 8, 68–78.

[24] ———, *Principles of distributed database systems*, Alan Apt, New Jersey, 1999.

[25] R. Ramakrishnan and J. Gehrke, *Database management systems*, McGra- Hill, Boston, 1998.

[26] K.-D. Schewe, *On the unification of query algebras and their extension to rational tree structures*, Proc. Australasian Database Conference (J. Roddick M. Orlowska, ed.), 2001.

[27] ———, *Database concepts*, Lecture Manuscrip, Massey University, Department of Information Systems, Palmerston North, New Zealand, 2002.

[28] ———, *Fragmentation of object oriented and semi-structured data*, Hele-MaiHaav, Proc. Baltic Conference on Databases and Information Systems, Kluwer Academic Publishers, 2002, pp. 1–14.

[29] K.-D. Schewe, B. Sridharan, M. Chrystall, S. Link, and H. Ma, *Database concepts*, Extramural Study and Administration Guide. Dept. of Information Systems, Massey University, Palmerston North, New Zealand, 2002.

[30] K.-D. Schewe and B. Thalheim, *Fundamental concepts of object oriented databases*, Acta Cybernetica **11** (1993), no. 4, 49–84.

[31] ———, *Principles of object oriented database design*, Information Modelling and Knowledge Bases (T. Kitahashi A. Márkus H. Jaakkola, H. Kangassalo, ed.), V. IOS Press, 1994, pp. 227–242.

[32] M. Stonebraker, L. A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, and P.Bernstein, *Third-generation database system manifesto*, ACM SIGMOD Record (1990).

[33] A. M. Tamhankar and S. Ram, *Database fragmentation and allocation: An integrated methodology and case study*, IEEE transactions on systems management **28** (1998), no. 3, 194–207.

[34] B. Thalheim, *Entity-relationship modeling*, Springer-Verlag, 2000.

[35] S. B. Yao, S. B. Navathe, and J-L. Weldon, *An integrated approach to database design*, Data Base Design Techniques I: Requirement and Logical Structures (New York), Springer-Verlag, 1982, Lecture Notes in Computer Science 132, pp. 1–30.